THE TAMIL NADU
Dr. AMBEDKAR LAW UNIVERSITY
CHENNAI

OBJECT ORIENTED PROGRAMMING LANGUAGE – JAVA AND WEB TECHOLOGY

COURSE MATERIAL FOR BCA.LL.B

(For the candidates admitted from academic year 2015 - 2016 onwards)

SCHOOL OF EXCELLENCE IN LAW

The study material deals with the concepts of object oriented programming - JAVA and web technology. The Java language's programming paradigm is based on the concept of OOP, which the language's features support. The Java language is a C-language derivative, so its syntax rules look much like C's. For example, code blocks are modularized into methods and delimited by braces ({ and }), and variables are declared before they are used. Structurally, the Java language starts with *packages*. A package is the Java language's namespace mechanism. Within packages are classes, and within classes are methods, variables, constants, and more.

Client-Side Mark-up and Scripting - Client-side technologies are things that operate in the browser. There is no need to interact with the server. These languages are generally very easy to use, and you can try them out right on your own computer.

HTML: Hypertext Mark-up Language - By now you surely know what HTML is! It is a basic mark-up language that you'll use to create the structure of your web pages. You can think of this as the framing for the house that you're building. It is the most basic and essential part of your web site - it gives your house shape, rooms, and structure.

JavaScript - JavaScript is a simple scripting language used to make things happen on your web page. In our home building analogy, JavaScript would be the hinges that make doors open and close and light switches turn on and off. It is important to recognize the difference between JavaScript and some of the server-side technologies discussed below. As a client-side language, JavaScript only works within the browser window. It cannot retrieve, create, or store data on the server; it can only manipulate things within the browser. Things that you might use JavaScript for include swapping images on mouseover, check to see if form fields have been completed, or making a web page redirect to another page.

Server-Side Programming Languages - There are many server-side programming languages that you can use on your web sites. These are languages that interact with the web server to manipulate data. In our home building analogy, server side programming would be all the functionality in the house: electrical wiring, plumbing, and duct work.

DEGREE OF BACHELOR OF COMPUTER APPLICATIONS & LAW (B.C.A.LL.B.,)

Title of the Course/ Paper	OBJECT ORIENTED PROGRAMMING LANGUAGE – JAVA AND WEB TECHOLOGY (CHD5A)		
Core – 1	III Year & Fifth Semester	Credit 4	
Objective of the course	This course introduces the details about the concepts of java (OOPs) and web techology		
Course outline	Unit 1 Object Oriented Programming Concepts – Objects – Classes – Methods and Messages – Abstraction and Encapsulation – Inheritance – Abstract Classes – Polymorphism – Objects and Classes in Java – Defining Classes – Methods – Method Overloading – Method Overriding – Access Specifiers – Static Members – Constructors – Finalize Method		
	Unit-2 Arrays – Strings – Packages – Java –Doc Comments –Inheritance – Class Hierarchy – Interfaces – Polymorphism – Dynamic Binding – Final Keyword – Abstract Classes – Exception Handling – Exception Hierarchy – Throwing and Catching Exceptions		
	Unit 3 Multi-Threaded Programming – Interrupting Threads – Thread States – Thread Properties – Thread Synchronization – Executors – Synchronizers – I/O Streams – Character and Byte Streams – Working with Files		
	Unit-4 An Introduction to HTML History – Versions –Some Fundamental HTML Elements – Relative Urls – Lists – Tables – Frames – Forms – The Java Script Language – History and Versions Introduction – Syntax – Variables and Data Types – Statements – Operators – Literals – Functions – Objects – Arrays – Built in Objects – Java Script Debuggers		
	Unit-5 Browers and the DOM – Introduction to the Document Object Model History and Levels – Intrinsic Event Handling – Modifying Element Style – The Document Tree – DOM Event Handling		

1 Recommended Texts

(i) Cay S Horstmann and Gray Cornell – Core Java Volume I Fundamentals
(ii) Jeffrey C Jackson – Web Technologies – A Computer Science Prespective

2 Reference Books

(i) K Arnold and J Gosling – The JAVA Programming Language
(ii) Timothy Budd – Understanding Object – Oriented Programming with Java
(iii) C Thomas Wu- An introduction to Object – Oriented Programming with Java

CONTENTS

CHAPTER 1

INTRODUCTION TO JAVA

Overview of Java

Java is one of the world's most important and widely used computer languages, and it has held this distinction for many years Unlike some other computer languages whose influence has waned with passage of time, while Java's has grown As of 2015, Java is one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers using and working on it.

Creation of Java

Java was developed by James Ghosling, Patrick Naughton, Mike Sheridan at Sun Microsystems Inc in 1991 It took 18 months to develop the first working version. The initial name was **Oak** but it was renamed to **Java** in 1995 as OAK was a registered trademark of another Tech company

Evolution of Java

Java was initially launched as Java 1 0 but soon after its initial release, Java 1.1 was launched. Java 1 1 redefined event handling, new library elements were added. In **Java 1.2** Swing and Collection framework was added and suspend(), resume() and stop() methods were deprecated from **Thread** class

No major changes were made into **Java 1.3** but the next release that was **Java 1.4** contained several important changes Keyword assert, chained exceptions and channel based I/O System was introduced

Java 1.5 was called **J2SE 5**, it added following major new features·

- Generics
- Annotations
- Enumerations
- For-each Loop
- Static Import
- Formatted I/O
- Concurrency utilities

Next major release was **Java SE 7** which included many new changes, like:

- Now **String** can be used to control Switch statement
- Multi Catch Exception
- *try-with-resource* statement
- Binary Integer Literals

- *Underscore* in numeric literals, etc

And the latest addition to the lot is, **Java SE 8**, it was released on March 18, 2014 Some of the major new features introduced in JAVA 8 are,

- Lambda Expressions
- New Collection Package java util stream to provide Stream API
- Enhanced Security
- Nashorn Javascript Engine included
- Parallel Array Sorting
- The JDBC-ODBC Bridge has been removed etc

Application of Java

Java is widely used in every corner of world and of human life Java is not only used in software's but is also widely used in designing hardware controlling software components There are more than 930 million JRE downloads each year and 3 billion mobile phones run java

Following are some other usage of Java

1. Developing Desktop Applications
2. Web Applications like Linkedin com, Snapdeal com etc
3. Mobile Operating System like Android
4. Embedded Systems
5. Robotics and games etc

Characteristics of Java – "white paper" Buzzords:

- **Object Oriented:** In Java, everything is an Object Java can be easily extended• since it is based on the Object model Platform Independent Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on

- **Simple:** Java is designed to be easy to learn If you understand the basic concept of OOP Java, it would be easy to master

- **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems Authentication techniques are based on public-key encryption

- **Architecture-neutral:** Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system

- **Portable:** Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset

- **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking

- **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously This design feature allows the developers to construct interactive applications that can run smoothly

- **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere The development process is more rapid and analytical since the linking is an incremental and light-weight process

- **High Performance:** With the use of Just-In-Time compilers, Java enables high performance

- **Distributed:** Java is designed for the distributed environment of the internet

- **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time

JDK:

Java Development Kit (JDK) is Kit which provides the environment to Develop and execute (run) the Java program For eg You (as Java Developer) are developing an accounting application on your machine, so what do you going to need into your machine to develop and run this desktop app? You are going to need J-D-K for that purpose for this you just need to go to official website of sun or oracle to download the latest version of JDK into your machine Hence, JDK is a kit(or package) which includes two things i) Development Tools (to provide an environment to develop your java programs) and ii) JRE (to execute your java program) JDK is only used by Java Developers

JRE:

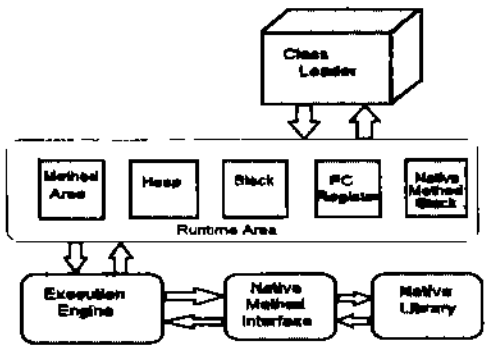
Java Runtime Environment (JRE) is an installation package which provides environment to only run the java program or application onto machine For eg(continuing with the same example) after developing accounting application , you want to run this application into client's

machine. Now, in this case client needs to run the application into his/her machine So, client should install JRE in order to run the application into his machine Hence, JRE is only used by them who only wants to run the Java Programs i e end users of your system.

JVM:

Java virtual Machine (JVM) is a virtual Machine that provides runtime environment to execute java byte code. The JVM doesn't understand Java typo, that's why you compile your * java files to obtain *.classfiles that contain the bytecodes understandable by the JVM JVM control execution of every Java program. It enables features such as automated exception handling, Garbage-collected heap

JVM Architecture



Class Loader. Class loader loads the Class for execution

Method area Stores pre-class structure as constant pool.

Heap Heap is in which objects are allocated

Stack Local variables and partial results are store here Each thread has a private JVM stack created when the thread is created.

Program register Program register holds the address of JVM instruction currently being executed

Native method stack. It contains all native used in application.

Executive Engine Execution engine controls execute of instructions contained in the methods of the classes

Native Method Interface Native method interface gives an interface between java code and native code during execution.

Native Method Libraries Native Libraries consist of files required for the execution of native code

Difference between C++ and Java

Java	C++
Java doesn't support Pointer concept	It support pointer concept
It doesn't support multiple inheritances.	It support multiple inheritance
Java does not include structures or unions.	It has structure and union concept
Java includes automatic garbage collection	C++ requires explicit memory management
Java has method overloading, but no operator overloading.	C++ supports both method overloading and operator overloading.
It is platform independent programming language	It is platform dependent programming language
It is mainly used for design web based application but also use for develop desktop application	It is used for design only desktop application like OS, Compiler etc
Java uses compiler and interpreter both	C++ use only Compiler.
Java is high level programming language in java we write code like simple English language	C++ is more nearer to hardware then Java

Review Questions

1 Define application software
2 Enlist the Buzzwords of java.
3 Describe the architecture of JVM?
4. Discuss the difference between java and C++.

CHAPTER 2

OBJECT ORIENTED PROGRAMMING

Definition of OOPs:

OOPs concepts in Java are the main ideas behind Java's Object Oriented Programming They are abstraction, encapsulation, inheritance, and polymorphism Grasping them is keys to understanding how Java works Basically, Java OOPs concepts let us create working methods and variables, then re-use all or part of them without compromising security

Abstraction:

Abstraction means using simple things to represent complexity We all know how to turn the TV on, but we don't need to know how it works in order to enjoy it In Java, abstraction means simple things like objects, classes, and variables represent more complex underlying code and data This is important because it lets avoid repeating the same work done multiple times

How Abstraction Works Abstraction works by letting programmers create useful, reusable tools For example, a programmer can create several different types of objects These can be variables, functions, or data structures Programmers can also create different classes of objects These are ways to define the objects For instance, a class of variable might be an address The class might specify that each address object shall have a name, street, city, and zip code The objects, in this case, might be employee addresses, customer addresses, or supplier addresses

Encapsulation:

This is the practice of keeping fields within a class private, then providing access to them via public methods It's a protective barrier that keeps the data and code safe within the class itself This way, we can re-use objects like code components or variables without allowing open access to the data system-wide

How Encapsulation Works Encapsulation lets us re-use functionality without jeopardizing security It's a powerful OOPs concept in Java because it helps us save a lot of time For example, we may create a piece of code that calls specific data from a database It may be useful to reuse that code with other databases or processes Encapsulation lets us do that while keeping our original data private It also lets us alter our original code without breaking it for others who have adopted it in the meantime

Inheritance:

This is a special feature of Object Oriented Programming in Java It lets programmers create new classes that share some of the attributes of existing classes This lets us build on previous work without reinventing the wheel

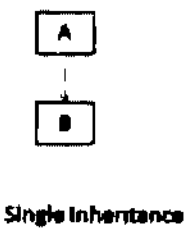
How Inheritance Works Inheritance is another labor-saving Java OOPs concept It works by letting a new class adopt the properties of another We call the inheriting class a **subclass** or a **child class** The original class is often called the **parent** We use the keyword **extends** to define a new class that inherits properties from an old class

Types of Inheritance:

- Single inheritance
- Multi-level inheritance
- Hierarchical inheritance
- Multiple inheritance
- Hybrid inheritance

Single inheritance

Single inheritance is damn easy to understand. When only one class extends from one base class only, then it is single inheritance The below flow diagram shows that class B extends only one class which is A Here A is a **parent class** of B and B would be **child class** of A



Single Inheritance

Example

```
Class A
{
  public void methodA()
  {
   System out.println("Base class method");
  }
}
Class B extends A
{
  public void methodB()
  {
```

```
    System.out println("Child class method"),
  }
  public static void main(String args[])
  {
    B obj = new B();
    obj methodA(), //calling super class method
    obj methodB(), //calling local method
  }
}
```

Multi-level inheritance

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A. For more details and example refer –Multilevel inheritance in Java.

Multilevel Inheritance

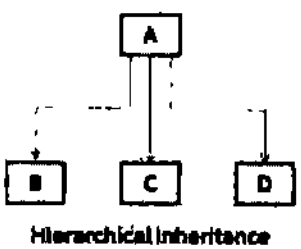
Example.

```
Class X
{
  public void methodX()
  {
    System.out println("Class X method"),
  }
}
Class Y extends X
{
public void methodY()
{
System out.println("class Y method"),
}
}
Class Z extends Y
{
  public void methodZ()
```

```
  {
   System out.println("class Z method"),
  }
  public static void main(String args[])
  {
   Z obj = new Z(),
   obj.methodX(), //calling grand parent class method
   obj.methodY(); //calling parent class method
   obj.methodZ(); //calling local method
  }
}
```

Hierarchical inheritance:

In "**Hierarchical inheritance**" one parent class will be inherited by many sub classes. As per the below example Class A will be inherited by Class B, Class C and Class D. Class A will be acting as a parent class for Class B, Class C and Class D



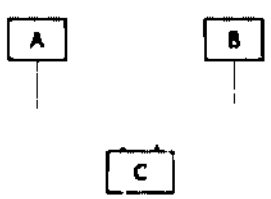
Hierarchical Inheritance

```
Class A
{
 public void methodA()
 {
   System out.println("method of Class A");
 }
}
Class B extends A
{
 public void methodB()
 {
   System out.println("method of Class B");
 }
}
Class C extends A
{
public void methodC()
{
System out.println("method of Class C");
}
```

```
}
Class D extends A
{
 public void methodD()
 {
   System out println("method of Class D"),
 }
}
Class MyClass
{
 public void methodB()
 {
   System out println("method of Class B"),
 }
 public static void main(String args[])
 {
   B obj1 = new B(),
   C obj2 = new C(),
   D obj3 = new D();
   obj1 methodA(),
   obj2 methodA(),
   obj3 methodA(),
 }
}
```

Multiple inheritance·

"**Multiple Inheritance**" refers to the concept of one class extending (Or inherits) from more than one base class The inheritance we learnt earlier had the concept of one base class or parent The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes



Multiple Inheritance

Note Multiple Inheritance is very rarely used in software projects Using Multiple inheritance often leads to problems in the hierarchy This results in unwanted complexity when further extending the class.

Why Java doesn't support multiple inheritance?

Multiple inheritance - java doesn't support it It is just to remove ambiguity, because multiple inheritance can cause ambiguity in few scenarios One of the most common scenario is Diamond problem Consider the above diagram which shows multiple inheritance as Class D extends both Class B & C Now lets assume we have a method in class A and class B & C overrides that method in their own way Here the problem comes – Because D is extending both B & C so if D wants to use the same method which method would be called (the overridden method of B or the overridden method of C) Ambiguity That's the main reason why Java doesn't support multiple inheritance

Multiple inheritance in Java can be achieved using interfaces

```
interface X
{
  public void myMethod(),
}
interface Y
{
  public void myMethod(),
}
class Demo implements X, Y
{
  public void myMethod()
  {
     System out println(" Multiple inheritance example using interfaces"),
  }
}
```

Hybrid inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Hierarchical** and **Multiple** inheritances A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be - Using interfaces By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.

```
interface A
{
   public void methodA(),
}
interface B extends A
{
   public void methodB(),
}
interface C extends A
{
   public void methodC(),
```

```
}
class D implements B, C
{
  public void methodA()
  {
     System.out println("MethodA");
  }
  public void methodB()
  {
     System out.println("MethodB");
  }
  public void methodC()
  {
     System out println("MethodC"),
  }
  public static void main(String args[])
  {
     D obj1= new D();
     obj1.methodA();
     obj1 methodB(),
     obj1 methodC(),
   }
}
```

Polymorphism:

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.

- It is a feature that allows one interface to be used for a general class of actions
- An operation may exhibit different behavior in different instances
- The behavior depends on the types of data used in the operation
- It plays an important role in allowing objects having different internal structures to share the same external interface
- Polymorphism is extensively used in implementing inheritance

Following concepts demonstrate different types of polymorphism in java.

1) Method Overloading is also known as Static and Compile time Polymorphism

2) Method Overriding is also known as Dynamic and Runtime Polymorphism

Method Definition:

A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name

Method Overloading:

In Java, it is possible to define two or more methods of same name in a class, provided that there argument list or parameters are different This concept is known as Method Overloading

1 To call an overloaded method in Java, it is must to use the type and/or number of arguments to determine which version of the overloaded method to actually call
2 Overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method
3 When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call
4 It allows the user to achieve compile time polymorphism
5 An overloaded method can throw different exceptions
6 It can have different access modifiers

Example

```
class Overload
{
   void demo (int a)
   {
     System out.println ("a  " + a),
   }
   void demo (int a, int b)
   {
     System.out println ("a and b  " + a + "," + b),
   }
   double demo(double a) {
     System out println("double a  " + a),
     return a*a,
   }
}
class MethodOverloading
{
   public static void main (String args [])
   {
      Overload Obj = new Overload(),
      double result,
      Obj  demo(10),
```

```
    Obj demo(10, 20),
    result = Obj demo(5 5),
    System out.println("O/P . " + result),
  }
}
```

Here the method demo() is overloaded 3 times first having 1 int parameter, second one has 2 int parameters and third one is having double arg. The methods are invoked or called with the same type and number of parameters used

Output:

```
a· 10
a and b 10,20
double a. 5 5
O/P 30 25
```

Rules for Method Overloading

1 Overloading can take place in the same class or in its sub-class
2. Constructor in Java can be overloaded
3 Overloaded methods must have a different argument list
4 Overloaded method ~~should always be the part of the same class~~ (can also take place in sub class), with same name but different parameters
5 The parameters may differ in their type or number, or in both
6 They may have the same or different return types
7. It is also known as compile time polymorphism

Method Overriding

Child class has the same method as of base class In such cases child class overrides the parent class method without even touching the source code of the base class This feature is known as method overriding.

Example

```
public class BaseClass
{
  public void methodToOverride() //Base class method
  {
    System out println ("I'm the method of BaseClass"),
  }
}
public class DerivedClass extends BaseClass
```

```
{
   public void methodToOverride() //Derived Class method
   {
      System out println ("I'm the method of DerivedClass");
   }
}

public class TestMethod
{
   public static void main (String args []) {
     // BaseClass reference and object
     BaseClass obj1 = new BaseClass();
     // BaseClass reference but DerivedClass object
     BaseClass obj2 = new DerivedClass();
     // Calls the method from BaseClass class
     obj1.methodToOverride();
     //Calls the method from DerivedClass class
     obj2.methodToOverride(),
   }
}
```

Output:

I'm the method of BaseClass
I'm the method of DerivedClass

Rules for Method Overriding.

1. applies only to inherited methods
2 object type (NOT reference variable type) determines which overridden method will be used at runtime
3. Overriding method can have different return type (refer this)
4 Overriding method must not have more restrictive access modifier
5 Abstract methods must be overridden
6 Static and final methods cannot be overridden
7 Constructors cannot be overridden
8 It is also known as Runtime polymorphism.

Review Questions

1. Why Java is object oriented?
2. Can multiple inheritance is possible in Java - Analyse
3. Rules for forming runtime polymorphism.
4. Why constructors are not overridden

CHAPTER 3

OBJECT AND CLASSES

The object-oriented design process involves the following three tasks

- dividing the problem domain into types of objects
- modeling the relationships between the types and
- modeling the attributes and behaviors of each type

These tasks are not listed in any particular order Most likely, you will perform these tasks iteratively throughout the design process In an object-oriented design, you identify the fundamental objects of the problem domain, the "things" involved You then classify the objects into types by identifying groups of objects that have common characteristics and behaviors The types of objects you identify in the problem domain become "types" in your solution The program you write will create and manipulate objects of these types By naming the types in your solution after the types in the problem, you build a vocabulary for expressing the solution out of the language you would use to describe the problem

In addition to types that correspond to elements in the problem, the "problem domain types," your solution will likely have types that don't correspond to anything in the problem domain For example, most programs will require types that deal with data management and user interface An example of a data-management type is a hash table You might use a hash table object in your program to speed lookup of a set of objects, even though there is no hash table object in the problem domain The objects you are looking up in the hash table, however, might represent objects that exist in the problem domain Some examples of user interface types might be button, window, and dialog

The fundamental task of abstraction in an object-oriented design is to identify objects in the problem domain and then to classify the objects into types As you divide the problem domain into types, you will to some degree model the relationships between the types as well Objects can have three kinds of relationships

- the has-a relationship
- the is-a relationship
- the uses-a relationship

The has-a relationship means that one type of object contains another or is composed of another Some examples are a car has-an engine, a bicycle has-a wheel, and a coffee cup has coffee The has-a relationship is modeled with composition

The is-a relationship means that one type of object is a more specific version of a general type Some examples are a car is-a vehicle, a bicycle is-a vehicle, and a coffee cup is-a cup The is-a relationship is modeled with inheritance

The uses-a relationship means that during some activity, one type of object uses another type of object Some examples are a car uses-a squeegee (during a window-washing activity), a bicycle uses-a pump (during a tire-pumping activity), and a coffee cup uses-a stirrer (during a stirring activity)

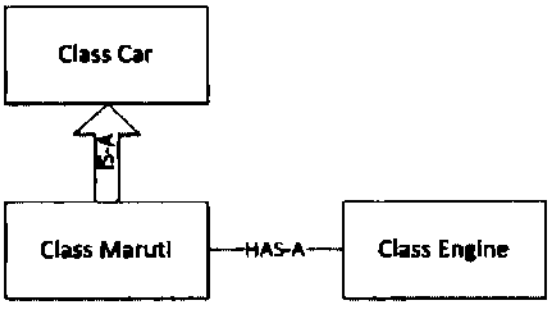
Along with dividing the problem domain into types and modeling their relationships, you must define attributes and behaviors that will characterize each type in the solution The attributes of a type define the nature of the state of objects of that type An object's state is composed of values for all the attributes of the type For example, two possible attributes for a bicycle type are speed and direction An object of type bicycle would therefore have a state that is composed of values for speed and direction Note that an object's state (the values of its attributes) can change over the lifetime of the object A bicycle object, for example, could have a state of 15 mph and north at one point in time Later, that same object could have state 10 mph and south In object-oriented thinking, interaction between objects is modeled as messages sent between objects and the action that objects take as a result When you model the behavior of a type you define a set of messages that objects of that type will accept, and the actions that objects of that type will take upon receipt of those messages The set of accepted messages and the resulting actions constitute services that are offered by the object

As the designer of a type, you decide what an object of that type will do when it receives a message Messages contain information, and an object can use the information contained in a received message along with the information represented by its own current state, to decide what to do It may do nothing It may send messages to other objects It may change its own state It may return some information to the message sender Or it may do all of these things In computer science circles, the term "message" is often associated with asynchronous messaging, in which received messages can queue up and be processed by the recipient at some later time In this object-oriented context, however, a message is simply a request coupled with some information that is passed to an object In general, an object begins to process a message immediately upon receipt and potentially returns a reply to the sender A message can have an effect that is delayed (similar to asynchronous messaging), but creating such a delayed effect is an option of the message recipient

These design activities are processes of abstraction because out of all the elements of the problem domain, you are selecting only those that are important As a result, in any one design you will likely ignore many elements of the problem domain In your solution, you won't model every type of object you can possibly identify in the problem domain, only those that matter to your solution Likewise, you won't model every attribute and every behavior of the types of objects you have chosen to represent in your

solution, just those attributes and behaviors that are important to your solution In a different problem domain, you might model different attributes and behaviors of the same types of objects. Thus, you are abstracting: pulling out what you feel is important about the problem domain, and using only those elements in your solution.

Let's understand these concepts with an example of Car class.




```
package relationships,
class Car {
  // Methods implementation and class/Instance members
  private String color;
  private int maxSpeed;
  public void carInfo(){
    System.out.println("Car Color= "+color + " Max Speed= " + maxSpeed),
  }
  public void setColor(String color) {
    this color = color;
  }
  public void setMaxSpeed(int maxSpeed) {
    this maxSpeed = maxSpeed;
  }
}
```

As shown above, Car class has a couple of instance variable and few methods. Maruti is a specific type of Car which extends Car class means Maruti IS-A Car.

```
class Maruti extends Car{
  //Maruti extends Car and thus inherits all methods from Car (except final and static)
  //Maruti can also define all its specific functionality
  public void MarutiStartDemo(){
    Engine MarutiEngine = new Engine(),
    MarutiEngine.start(),
    }
  }
```

Maruti class uses Engine object's start() method via composition We can say that Maruti class HAS-A Engine

```
package relationships,
public class Engine {
  public void start(){
    System out println("Engine Started "),
  }
  public void stop(){
    System out.println("Engine Stopped "),
  }
}
```

RelationsDemo class is making object of Maruti class and initialized it Though Maruti class does not have setColor(), setMaxSpeed() and carInfo() methods still we can use it due to IS-A relationship of Maruti class with Car class

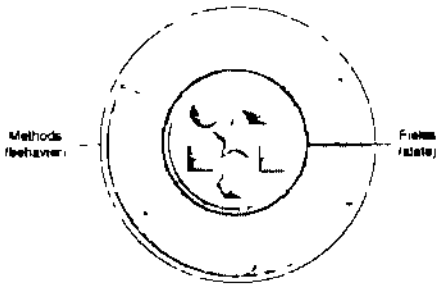
```
package relationships,
public class RelationsDemo {
  public static void main(String[] args) {
    Maruti myMaruti = new Maruti(),
    myMaruti setColor("RED"),
    myMaruti setMaxSpeed(180),
    myMaruti carInfo(),
    myMaruti MarutiStartDemo(),
  }
}
```

What Is an Object?

Objects are keys to understand object-oriented technology Look around right now and you'll find many examples of real-world objects your dog, your desk, your television set, your bicycle Real-world objects share two characteristics· They all have state and behavior Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes) Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions "What possible states can this object be in?" and "What possible behavior can this object perform?" Make sure to write down your observations As you do, you'll notice that real-world objects vary in complexity, your desktop lamp may have only two

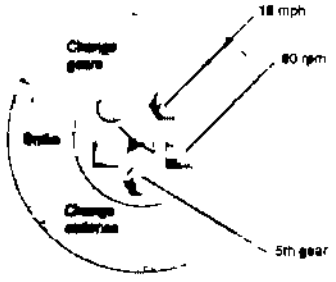
possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune) You may also notice that some objects, in turn, will also contain other objects These real-world observations all translate into the world of object-oriented programming A circle with an inner circle filled with items, surrounded by gray wedges representing methods that allow access to the inner circle



A software object

Software objects are conceptually similar to real-world objects they too consist of state and related behavior An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages) Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming

Consider a bicycle, for example



A picture of an object, with bicycle methods and instance variables

A bicycle modeled as a software object By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6

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

Modularity The source code for an object can be written and maintained independently of the source code for other objects Once created, an object can be easily passed around inside the system

Information-hiding By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world

Code re-use If an object already exists (perhaps written by another software developer), you can use that object in your program This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code

Plug ability and debugging ease If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement This is analogous to fixing mechanical problems in the real world If a bolt breaks, you replace it, not the entire machine

Why Use Objects?

- They model the real world combination of behavior and state
- Behavior may just wrap access to state (personnel record) be very complex (printer)
- They make programming easier (i e) Reduced complexity and easy maintenance

Difference between object and class

There are many differences between object and class A list of differences between object and class are given below

No.	Object	Class
1)	Object is an **instance** of a class	Class is a **blueprint or template** from which objects are created
2)	Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc	Class is a **group of similar objects.**
3)	Object is a **physical** entity	Class is a **logical** entity
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student(),	Class is declared using **class keyword** e.g class Student{}
5)	Object is created **many times** as per requirement	Class is declared **once**
6)	Object **allocates memory when it is created**	Class **doesn't allocated memory when it is**

		created.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword

METHODS AND METHOD DECLARATION IN JAVA:

A method in Java is a block of statements that has a name and can be executed by calling (also called invoking) it from some other place in your program. Along with fields, methods are one of the two elements that are considered members of a class. (Constructors and initializes are not considered class members) Every program must have at least one method for the program to accomplish any work And every program must have a method named main, which is the method first invoked when the program is run All methods — including the main method --- must begin with a method declaration Here's the basic form of a method declaration

visibility [static] return-type method-name (parameter-list)

{

statements. .

}

The following list describes the method declaration

- visibility: The visibility of a method determines whether the method is available to other classes. The options are
  - public· Allows any other class to access the method
  - private· Hides the method from other classes
  - protected· Lets subclasses use the method but hides the method from other classes

- static This optional keyword declares that the method is a static method, which means that you can call it without first creating an instance of the class in which it's defined. The main method must always be static, and any other methods in the class that contains the main method usually should be static as well
- return-type After the word static comes the return type, which indicates whether the method returns a value when it is called — and if so, what type the value is If the method doesn't return a value, specify void If you specify a return type other than void, the method must end with a return statement that returns a value of the correct type Like int, float etc.

- method-name Now comes the name of your method The rules for making up method names are the same as the rules for creating other identifiers Use any combination of letters and numbers, but start with a letter.
- parameter list: You can pass one or more values to a method by listing the values in parentheses following the method name The parameter list in the method declaration lets Java know what types of parameters a method should expect to receive and provides names so that the statements in the method's body can access the parameters as local variables.

> If the method doesn't accept parameters, you must still code the parentheses that surround the parameter list. You just leave the parentheses empty

- statements. One or more Java statements that comprise the method body, enclosed in a set of braces. Unlike Java statements such as if, while, and for, the method body requires you to use the braces even if the body consists of only one statement.

Review Questions

1 What are the function visibilities explain?
2 Why we need main function for executing the program?
3 State the difference between objects and class.
4. Explain class relationships in detail.

CHAPTER 4

CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS

Constructor is a block of code that initializes the newly created object A constructor resembles an instance method in java but it's not a method as it doesn't have a return type In short, constructor and methods are different Constructor has same name as the class name

Example

```
public class MyClass {
  //This is the constructor
  MyClass() {
  }

}
```

How does a constructor work

To understand the working of constructor, lets take an example lets say we have a class MyClass When we create the object of MyClass like this

MyClass obj = new MyClass()

The **new keyword** here creates the object of class MyClass and invokes the constructor to initialize this newly created object

A simple constructor program in java

Here we have created an object obj of class Hello and then we displayed the instance variable name of the object which is what we have passed to the name during initialization in constructor This shows that when we created the object obj the constructor got invoked (Refer this keyword in this chapter end)

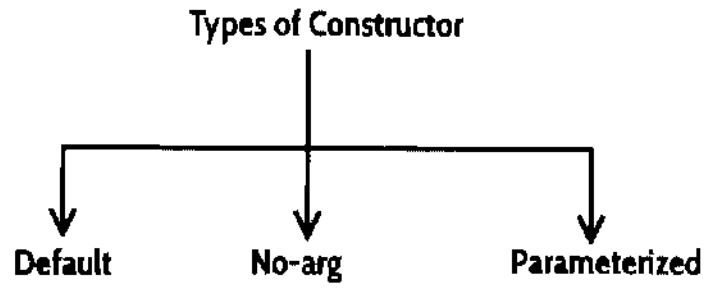
```
public class Hello {
  String name;
  //Constructor
  Hello() {
    this name = "Java";
  }
  public static void main(String[] args) {
    Hello obj = new Hello();
    System out println(obj name);
  }
}
```

Output:

Java

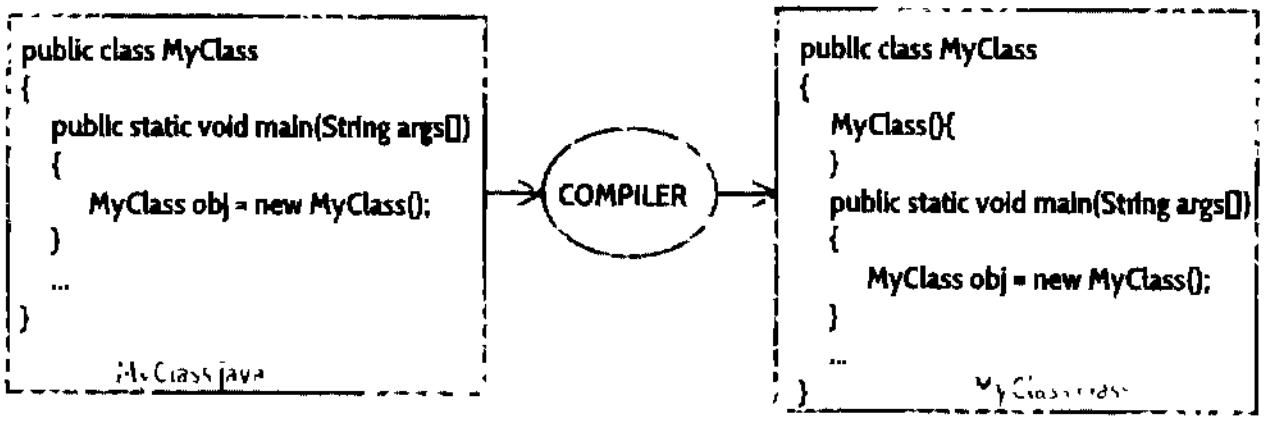
Types of Constructors

There are three types of constructors Default, No-arg constructor and Parameterized



Default constructor

If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code on your behalf This constructor is known as default constructor You would not find it in your source code (the java file) as it would be inserted into the code during compilation and exists in class file If you implement any constructor then you no longer receive a default constructor from Java compiler This process is shown in the diagram below



no-argument constructor:

Constructor with no arguments is known as **no-argument constructor** The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty Although you may see some people claim that that default and no-arg constructor is same but in fact they are not, even if you write public Demo() { } in your class Demo it cannot be called default constructor since you have written the code of it

Example no-arg constructor

```
class Demo
{
   public Demo()
```

```
    {
      System out println("This is a no argument constructor"),
    }
    public static void main(String args[]) {
        new Demo(),
    }
}
```

Output

This is a no argument constructor

Parameterized constructor:

Constructor with arguments(or you can say parameters) is known as Parameterized constructor

Example: parameterized constructor

In this example we have a parameterized constructor with two parameters id and name. While creating the objects obj1 and obj2 I have passed two arguments so that this constructor gets invoked after creation of obj1 and obj2

```
public class Employee {
  int empId,
  String empName;
  //parameterized constructor with two parameters
  Employee(int id, String name){
     this empId = id,
     this empName = name;
  }
  void info(){
     System out.println("Id  "+empId+" Name  "+empName),
  }

  public static void main(String args[]){
        Employee obj1 = new Employee(10245,"Chaitanya"),
        Employee obj2 = new Employee(92232,"Negan"),
        obj1 info(),
        obj2.info();
  }
}
```

Output:

Id: 10245 Name: Chaitanya
Id. 92232 Name: Negan

Example2 parameterized constructor

In this example, we have two constructors, a default constructor and a parameterized constructor When we do not pass any parameter while creating the object using new keyword then default constructor is invoked, however when you pass a parameter then parameterized constructor that matches with the passed parameters list gets invoked.

```
class Example2
{
    private int var;
    //default constructor
    public Example2()
    {
        this.var = 10,
    }
    //parameterized constructor
    public Example2(int num)
    {
        this var = num,
    }
    public int getValue()
    {
         return var,
    }
    public static void main(String args[])
    {
         Example2 obj = new Example2(),
         Example2 obj2 = new Example2(100);
         System out.println("var is. "+obj.getValue());
         System out.println("var is  "+obj2.getValue()),
    }
}
```

Output:

```
var is· 10
var is. 100
```

What if you implement only parameterized constructor in class

```
class Example3
{
```

```
    private int var;
    public Example3(int num)
    {
        var=num;
    }
    public int getValue()
    {
        return var;
    }
    public static void main(String args[])
    {
        Example3 myobj = new Example3();
        System out println("value of var is  "+myobj getValue());
    }
}
```

Output It will throw a compilation error The reason is, the statement Example3 myobj = new

Example3() is invoking a default constructor which we don't have in our program When you don't implement any constructor in your class, compiler inserts the default constructor into your code, however when you implement any constructor (in above example I have implemented parameterized constructor with int parameter), then you don't receive the default constructor by compiler into your code If we remove the parameterized constructor from the above code then the program would run fine, because then compiler would insert the default constructor into your code

Constructor Chaining:

When a constructor calls another constructor of same class then this is called constructor chaining Constructor chaining can be done in two ways

- Within same class It can be done using this() keyword for constructors in same class
- From base class by using super() keyword to call constructor from the base class

Constructor chaining occurs through inheritance A sub class constructor's task is to call super class's constructor first This ensures that creation of sub class's object starts with the initialization of the data members of the super class There could be any numbers of classes in inheritance chain Every constructor calls up the chain till class at the top is reached

Why do we need constructor chaining?

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable

Constructor Chaining within same class using this() keyword

```
// Java program to illustrate Constructor Chaining within same class
// Using this() keyword
class Temp
{
  // default constructor 1
  // default constructor will call another constructor
  // using this keyword from same class
  Temp()
  {
    // calls constructor 2
    this(5),
    System.out.println("The Default constructor"),
  }

  // parameterized constructor 2
  Temp(int x)
  {
    // calls constructor 3
    this(5, 15);
    System.out println(x),
  }

  // parameterized constructor 3
  Temp(int x, int y)
  {
    System out println(x * y),
  }

  public static void main(String args[])
  {
    // invokes default constructor first
    new Temp();
  }
}
```

Output.

The Default constructor

5

75

Rules of constructor chaining

1 The this() expression should always be the first line of the constructor.

2 There should be at-least be one constructor without the this() keyword (constructor 3 in above example)

3 Constructor chaining can be achieved in any order

Constructor Chaining to other class using super() keyword

Whenever a child class constructor gets invoked it implicitly invokes the constructor of parent class You can also say that the compiler inserts a super(),statement at the beginning of child class constructor

```
// Java program to illustrate Constructor Chaining to other class using super() keyword
class Base
{
  String name,
  // constructor 1
  Base()
  {
    this(""),
    System out println("No-argument constructor of" + " base class"),
  }
  // constructor 2
  Base(String name)
  {
    this name = name,
    System out println("Calling parameterized constructor"+ " of base"),
  }
}
class Derived extends Base
{
  // constructor 3
  Derived()
  {
    System out println("No-argument constructor " + "of derived"),
  }
  // parameterized constructor 4
  Derived(String name)
  {
    // invokes base class constructor 2
    super(name),
    System out println("Calling parameterized " + "constructor of derived"),
  }
  public static void main(String args[])  {
    // calls parameterized constructor 4
    Derived obj = new Derived("test"),
     // Calls No-argument constructor Derived obj = new Derived(),
  }
}
```

Difference between Constructor and Method:

1 **Constructor** is used to initialize the state of object, where as **method** is expose the behavior of object
2 **Constructor** must not have return type where as **method** must have return type
3 **Constructor** name same as the class name where as **method** may not the same class name
4 **Constructor** invoke implicitly where as **method** invoke explicitly
5 **Constructor** compiler provide default constructor where as **method** compiler doesn't provide

Destructor

A destructor is a special method called automatically during the destruction of an object Actions executed in the destructor include the following

- Recovering the heap space allocated during the lifetime of an object
- Closing file or database connections
- Releasing network resources
- Releasing resource locks
- Other housekeeping tasks

Destructors are called explicitly in C++ However, in Java this is not the case, as the allocation and release of memory allocated to objects are implicitly handled by the garbage collector While destructors in Java (called finalizers) are nondeterministic. However, Java finalizers have to be explicitly invoked since their invocation is not guaranteed

Key properties of destructors can be summarized as follows

- Automatic invocation and no explicit call from user code
- Overloading or inheritance not allowed.
- Access modifiers or parameters not to be specified
- Order of call to destructor in a derived class is from the most derived to the least derived class
- Called not only during the object destruction, but also when the object instance is no longer eligible for access
- Used only to release expensive unmanaged resources (like windows, network connection, etc) that the object holds, rather than for releasing managed references

/* Example of destructor - This is a very simple class that I use to demonstrate the scheduling and running of destructor methods - looking at the static class member that counts up and down as objects are created and destroyed */

```
public class Thing {
public static int number_of_things = 0,
public String what,
public Thing (String what) {
    this what = what,
    number_of_things++,
}
```

```
protected void finalize () {
    number_of_things--;
    }
}
```

Garbage Collector

A garbage collector is a piece of software that performs automatic memory management Its job is to free any unused memory and ensure that no memory is freed while it is still in use Some languages such as Java and NET languages feature automatic garbage collection, whereas others such as C/C++ require the programmer to manual manage memory Garbage collection was first introduced by Lisp creator John McCarthy to ease the manual memory management when working with the Lisp language

The three main techniques used by a garbage collector to perform automatic memory management are as follows

- Reference counting -- The reference to each object is counted using a counter variable When the counter reaches zero, it denotes that the object is no longer needed and thus is recycled
- Mark and sweep -- A recursive traversal of all reachable objects is carried out on all data regions, and reachable objects are marked The unmarked objects are then recycled
- Stop and copy – The memory heap is divided into two sections a section that contains the objects and an empty section where the objects are transferred (copied) if found to be marked The unmarked objects in the first section are recycled by emptying it

When a block of memory assigned to a pointer/object has been freed, the pointer/object must be reset to a null value, otherwise, it is dangling, i e , pointing to an invalid memory block Garbage collection helps reduce bugs and security risks caused by dangling pointers and memory leak problems The disadvantages of using a garbage collector include the extra overhead on resources and performance Running a garbage collector may also slow down the system and thus decrease its performance

Review Questions

1 Can we have static constructors in Java?
2 Difference between Constructors and Method
3 Explain garbage collector
4 How destructors are used in Java?

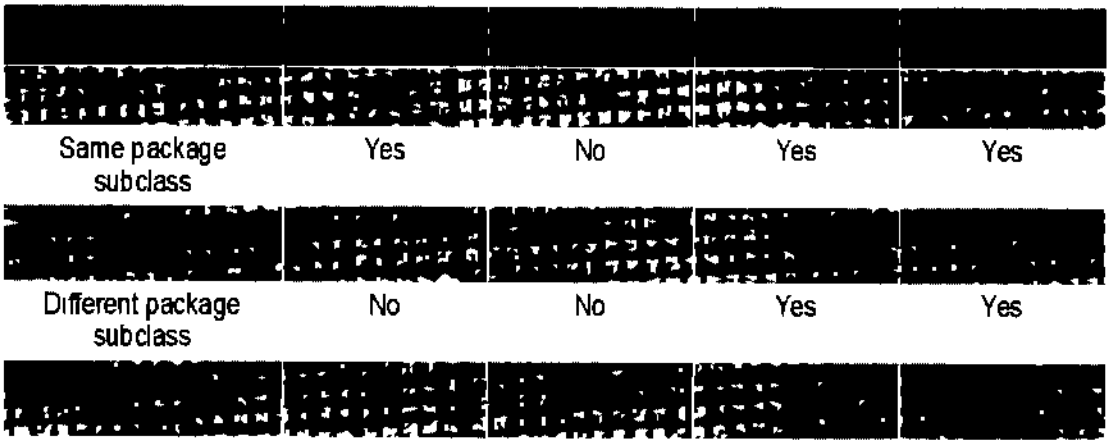
CHAPTER 5

ACCESS MODIFIERS AND PACKAGES

Access Modifiers:

Access modifiers in Java help to restrict the scope of a class, constructor, variable, method or data member There are four types of access modifiers available in java

1 Default – No keyword required
2 Private
3 Protected
4 Public



Default

- When no access modifier is specified for a class, method or data member – It is said to be having the **default** access modifier by default
- The data members, class or methods which are not declared using any access modifiers i e having default access modifier are accessible **only within the same package**
- In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of second package

```
//Java program to illustrate default modifier
package p1, //Class Example1 is having Default access modifier
class Example1 {
  void display() {
      System out println("Hello World!"),     }}
//Java program to illustrate error while using class from different package with default modifier
package p2,
import p1 *,//This class is having default access modifier
class Example2{
  public static void main(String args[])     {
      //accessing class Example1 from package p1
      Example1 obj = new Example1(),        obj display(),     }}
```

Output Compile time error

Private

- The private access modifier is specified using the keyword **private**
- The methods or data members declared as private are accessible only **within the class** in which they are declared
- Any other **class of same package will not be able to access** these members
- Classes or interface cannot be declared as private

In this example, we will create two classes A and B within same package p1 We will declare a method in class A as private and try to access this method from class B and see the result

```
//Java program to illustrate error while using class from different package with private modifier
package p1,
class A{
 private void display()   {
    System out println("hello world"),   }}
class B{
 public static void main(String args[])     {
      A obj = new A(),
      //trying to access private method of another class
      obj display(),     }}
```

Output

error display() has private access in A so, it cannot be used

Protected

- The protected access modifier is specified using the keyword **protected**
- The methods or data members declared as protected are **accessible within same package or sub classes in different package.**

In this example, we will create two packages p1 and p2 Class A in p1 is made public, to access it in p2 The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B

```
//Java program to illustrate protected modifier
package p1, //Class A
public class A{
  protected void display()   {
     System out println("Hello world"),   }}
 Java program to illustrate protected modifier
package p2,
import p1 *,
//importing all classes in package p1Class B is subclass of A
class B extends A{
  public static void main(String args[])   {
    B obj = new B(),        obj display(),
} }
```

Public

- The public access modifier is specified using the keyword **public**
- The public access modifier has the **widest scope** among all other access modifiers
- Classes, methods or data members, which are declared as public are **accessible from everywhere** in the program
- There is no restriction on the scope of public data members.

```
//Java program to illustrate public modifier
package p1,
public class A{
  public void display()    {
      System out println("Hello world"),    }}
package p2,
import p1 *,
class B{
   public static void main(String args[])    {
       A obj = new A,
       obj display(),    }}
```

Packages:

Package in Java is a mechanism to encapsulate a group of classes, interfaces and sub packages Many implementations of Java use a hierarchical file system to manage source and class files It is easy to organize class files into packages All we need to do is put related class files in the same directory, give the directory a name that relates to the purpose of the classes, and add a line to the top of each class file that declares the package name, which is the same as the directory name where they reside In java there are already many predefined packages that we use while programming

For example java lang, java io, java util etc
However one of the most useful feature of java is that we can define our own packages

Advantages of using a package

- Reusability Reusability of code is one of the most important requirements in the software industry Reusability saves time, effort and also ensures consistency A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program
- Easy to locate the files
- In real life situation there may arise scenarios where we need to define files of the same name This may lead to "name-space collisions" Packages are a way of avoiding "name-space collisions"

Types of package:

1) User defined package The package we create is called user-defined package
2) Built-in package The already defined package like java io *, java lang * etc are known as built-in packages

A *package* is a collection of classes and interfaces Each package has its own name and organizes its top-level (that is; non-nested) classes and interfaces into a separate *namespace*, or name collection Although same-named classes and interfaces cannot appear in the same package, they can appear in different packages because a separate namespace assigns to each package

From an implementation perspective, equating a package with a directory proves helpful, as does equating a package's classes and interfaces with a directory's classfiles Keep in mind other approaches such as the use of databases to implementing packages, so do not get into the habit of always equating packages with directories But because many JVMs use directories to implement packages, this article equates packages with directories The Java 2 SDK organizes its vast collection of classes and interfaces into a tree-like hierarchy of packages within packages, which is equivalent to directories within directories Examples of Java's packages include

- **java.lang** A collection of language-related classes, such as Object and String, organized in the java package's lang subpackage
- **java.lang.ref** A collection of reference-related language classes, such as SoftReference and ReferenceQueue, organized in the ref sub-subpackage of the java package's lang subpackage
- **javax.swing** A collection of Swing-related component classes, such as JButton, and interfaces, such as ButtonModel, organized in the javaxpackage's swing subpackage

Period characters separate package names For example, in javax swing, a period character separates package name javax from subpackage name swing A period character is the platform-independent equivalent of forward slash characters (/), backslash characters (\), or other characters to separate directory names in a directory-based package implementation, database branches in a hierarchical database-based package implementation, and so on

Create a package of classes and interfaces

Every source file's classes and interfaces organize into a package In the package directive's absence, those classes and interfaces belong to the unnamed package (the directory the JVM regards as the current directory—the directory where a Java program begins its execution via the Windows java exe, or OS-equivalent, program—and contains no subpackages) But if the package directive appears in a source file, that directive names the package for those classes and interfaces Use the following syntax to specify a package directive in source code

'package' packageName ['.' subpackageName] ';'

A package directive begins with the package keyword An identifier that names a package, packageName, immediately follows If classes and interfaces are to appear in a subpackage (at some level) within packageName, one or more period-separated subpackageName identifiers appear after packageName

The following code fragment presents a pair of package directives

```
package game;
package game.devices;
```

The first package directive identifies a package named game All classes and interfaces appearing in that directive's source file organize in the game package The second package directive identifies a subpackage named devices, which resides in a package named game. All classes and interfaces appearing in that directive's source file organize in the game package's devices subpackage If a JVM implementation maps package names to directory names, game devicesmaps to a game\devices directory hierarchy under Windows and a game/devicesdirectory hierarchy under Linux or Solaris

Only one package directive can appear in a source file Furthermore, the package directive must be the first code (apart from comments) in that file Violating either rule causes Java's compiler to report an error

```
// A.java
package testpkg,
public class A{
  int x = 1;
  public int y = 2,  protected int z = 3,
  int returnx ()  {
    return x,  }
  public int returny ()  {
    return y,  }
  protected int returnz ()  {
    return z,  }
  public interface StartStop  {
    void start (),    void stop (),  }}
class B{
  public static void hello ()  {
    System out println ("hello");}}
```

Review Questions

1 Why we need user-defined packages?
2 Explain access modifiers in detail
3 How built-in packages are used in Java?
4 Discuss the advantage of protected and public modifiers

CHAPTER 6

ARRAYS AND STRINGS

Concept of Array:

An array is a collection of similar data types Array is container object that hold values of homogenous type It is also known as static data structure because size of an array must be specified at the time of its declaration An array can be either primitive or reference type It gets memory in heap area Index of array starts from zero to size - 1

Features of Array

- It is always indexed Index begins from 0
- It is a collection of similar data types
- It occupies a contiguous memory location

Array Declaration

Syntax :

datatype[ ] *identifier*, or **datatype** identifier[ ],

Both are valid syntax for array declaration But the former is more readable

Example :

```
int[ ] arr,
char[ ] arr,
short[ ] arr,
long[ ] arr,
int[ ][ ] arr,  // two dimensional array
```

Initialization of Array

new operator is used to initialize an array

Example :

```
int[ ] arr = new int[10],   //this creates an empty array named arr of integer type whose size is 10
or
int[ ] arr = {10,20,30,40,50},  //this creates an array named arr whose elements are given
```

Accessing array element

As mention earlier array index starts from 0 To access nth element of an array Syntax

arrayname[n-1],

Example To access 4th element of a given array

```
int[ ] arr = {10,20,30,40},
System out println("Element at 4th place" + arr[3]),
```

The above code will print the 4th element of array arr on console

Note: To find the length of an array, we can use the following syntax array_name length There are no braces in front of length It's not length()

For each or enhanced for loop

J2SE 5 introduces special type of for loop called foi each loop to access elements of array Using for each loop you can access complete array sequentially without using index of array Let us see an example of for each loop

```
class Test{
public static void main(String[] args) {
  int[] arr = {10, 20, 30, 40},
       for(int x  arr) {
              System out println(x), }
 }}
```

Output :

10
20
30
40

Multi-Dimensional Array

A multi-dimensional array is very much similar to a single dimensional array It can have multiple rows and multiple columns unlike single dimensional array, which can have only one full row or one full column

Array Declaration Syntax:

datatype[ ][ ] identifier, or datatype identifier[ ][ ],

Initialization of Array

new operator is used to initialize an array

Example:

```
int[ ][ ] arr = new int[10][10],   //10 by 10 is the size of array  or
int[ ][ ] arr = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}},
// 3 by 5 is the size of the array
```

Accessing array element

For both, row and column, the index begins from 0

Syntax:

array_name[m-1][n-1]

Example:

```
int arr[ ][ ] = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}},
System out println("Element at (2,3) place" + arr[1][2]),
```

Jagged Array

Jagged means to have an uneven edge or surface In java, a jagged array means to have a multi-dimensional array with uneven size of rows in it

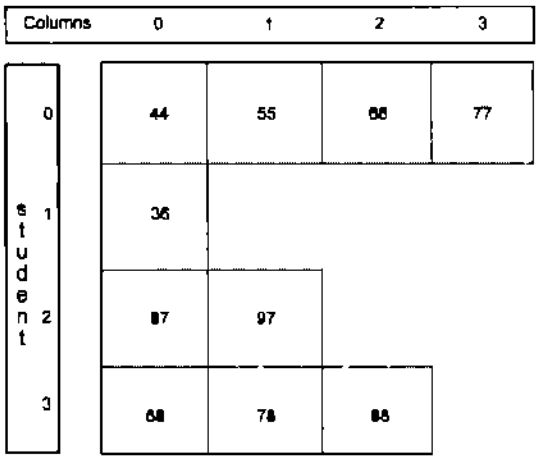


Figure Varying columns 2D array - matrix form

Initialization of Jagged Array

new operator is used to initialize an array

Example:

int[ ][ ] arr = new int[3][ ], //there will be 10 arrays whose size is variable
arr[0] = new int[3], arr[1] = new int[4], arr[2] = new int[5],

Advantage of Array

- Code Optimization It makes the code optimized, we can retrieve or sort the data easily
- Random access We can get any data located at any index position

Disadvantage of Array

- Size Limit We can store only fixed size of elements in the array It doesn't grow its size at runtime To solve this problem, collection framework is used in java

String:

String is nothing but a sequence of characters, for e g "Hello" is a string of 5 characters In java, string is an immutable object which means it is constant and cannot be changed once it has been created

Creating a String:

There are two ways to create a String in Java

1 String literal
2 Using new keyword

String literal

In java, Strings can be created like this Assigning a String literal to a String instance

String str1 = "Welcome";
String str2 = "Welcome";

The problem with this approach As stated in the beginning that String is an object in Java However we have not created any string object using new keyword above The compiler does that task for us, it creates a string object having the string literal (that we have provided , in this case it is "Welcome") and assigns it to the provided string instances **But** if the object already exist in the memory it does not create a new Object rather it assigns the same old object to the new instance, that means even though we have two string instances above(str1 and str2) compiler only created on string object (having the value "Welcome") and assigned the same to both the instances For example there are 10 string instances that have same value, it means in memory there is only one object having the value and all the 10 string instances would be pointing to the same object What if we want to have two different objects with the same string? For that we would need to create strings using **new keyword**

Using New Keyword:

As mentioned above that when we tried to assign the same string object to two different literals, compiler only created one object and made both of the literals to point the same object To overcome that approach we can create strings like this

String str1 = new String("Welcome"),
String str2 = new String("Welcome");
In this case compiler would create two different object in memory having the same text

Strings Initialization:

There are two ways to declare a string in Java, and the most common way to create a string is to write Example

String name = "Alex",

Through char array

```
public class Sample {

   public static void main(String args[]) {
      char[] nameArray = {'A', 'l', 'e', 'x'},
      String name = new String(nameArray),
      System out println(name),
   }
}
```

String Methods:

char charAt(int index): It returns the character at the specified index. Specified index value should be between 0 to length() -1 both inclusive It throws IndexOutOfBoundsException if index<0||>= length of String

int codePointAt(int index): It is similar to the charAt method however it returns the Uni code point value of specified index rather than the character itself

void getChars(int srcBegin, int srcEnd, char[] dest, int destBegin) It copies the characters of **src** array to the **dest** array Only the specified range is being copied (srcBegin to srcEnd) to the dest subarray(starting fromdestBegin)

boolean equals(Object obj): Compares the string with the specified string and returns true if both matches else false

boolean contentEquals(StringBuffer sb): It compares the string to the specified string buffer

boolean equalsIgnoreCase(String string): It works same as equals method but it doesn't consider the case while comparing strings It does a case insensitive comparison

int compareTo(String string): This method compares the two strings based on the Unicode value of each character in the strings

int compareToIgnoreCase(String string) Same as CompareTo method however it ignores the case during comparison

boolean regionMatches(int srcoffset, String dest, int destoffset, int len): It compares the substring of input to the substring of specified string

boolean regionMatches(boolean ignoreCase, int srcoffset, String dest, int destoffset, int len) Another variation of regionMatches method with the extra boolean argument to specify whether the comparison is case sensitive or case insensitive

boolean startsWith(String prefix, int offset): It checks whether the substring (starting from the specified offset index) is having the specified prefix or not

boolean startsWith(String prefix): It tests whether the string is having specified prefix, if yes then it returns true else false

boolean endsWith(String suffix): Checks whether the string ends with the specified suffix

int hashCode(): It returns the hash code of the string

int indexOf(int ch): Returns the index of first occurrence of the specified character ch in the string

int indexOf(int ch, int fromIndex): Same as indexOf method however it starts searching in the string from the specified fromIndex

int lastIndexOf(int ch) It returns the last occurrence of the character ch in the string

int lastIndexOf(int ch, int fromIndex): Same as lastIndexOf(int ch) method, it starts search from fromIndex

int indexOf(String str): This method returns the index of first occurrence of specified substring str.

int lastindexOf(String str): Returns the index of last occurrence of string str

String substring(int beginIndex): It returns the substring of the string The substring starts with the character at the specified index

String substring(int beginIndex, int endIndex): Returns the substring The substring starts with character at beginIndex and ends with the character at endIndex

String concat(String str): Concatenates the specified string "str" at the end of the string

String replace(char oldChar, char newChar): It returns the new updated string after changing all the occurrences of oldChar with the newChar

boolean contains(CharSequence s) It checks whether the string contains the specified sequence of char values If yes then it returns true else false It throws NullPointerException of 's' is null

String replaceFirst(String regex, String replacement) It replaces the first occurrence of substring that fits the given regular expression "regex" with the specified replacement string

String replaceAll(String regex, String replacement): It replaces all the occurrences of substrings that fits the regular expression regex with the replacement string

String[] split(String regex, int limit): It splits the string and returns the array of substrings that matches the given regular expression limit is a result threshold here

String[] split(String regex): Same as split(String regex, int limit) method however it does not have any threshold limit

String toLowerCase(Locale locale): It converts the string to lower case string using the rules defined by given locale

String toLowerCase(): Equivalent to toLowerCase(Locale getDefault())

String toUpperCase(Locale locale): Converts the string to upper case string using the rules defined by specified locale

String toUpperCase(): Equivalent to toUpperCase(Locale getDefault())

String trim(): Returns the substring after omitting leading and trailing white spaces from the original string

char[] toCharArray(): Converts the string to a character array

static String copyValueOf(char[] data): It returns a string that contains the characters of the specified character array

static String copyValueOf(char[] data, int offset, int count): Same as above method with two extra arguments – initial offset of subarray and length of subarray

static String valueOf(data type): This method returns a string representation of specified data type

byte[] getBytes(String charsetName): It converts the String into sequence of bytes using the specified charset encoding and returns the array of resulted bytes

byte[] getBytes(): This method is similar to the above method it just uses the default charset encoding for converting the string into sequence of bytes

int length(): It returns the length of a String

boolean matches(String regex): It checks whether the String is matching with the specified regular expression regex

Review Questions

1 Why we String are called as objects in java?
2 Explain any six strings functions with example each
3 How multi-dimensional arrays are implemented? Explain with example
4 Discuss advantage and disadvantage of using array

CHAPTER 7

IDENTIFIERS, DATA TYPES AND VARIABLES

Identifiers:

In programming languages, identifiers are used for identification purpose In Java an identifier can be a class name, method name, variable name or a label

For example·

```
public class Test
{
        public static void main(String[] args)
        {
                int a = 20,
        }
}
```

In the above java code, we have 5 identifiers namely·

- **Test** class name
- **main**: method name
- **String**· predefined class name
- **args** variable name.
- **a** variable name

Rules for defining Identifiers

There are certain rules for defining valid identifiers These rules must be followed, otherwise we get compile-time error

- The only allowed characters for identifiers are all alphanumeric characters([**A-Z**],[**a-z**],[**0-9**]), '**$**'(dollar sign) and '_' (underscore) For example "geek@" is not a valid java identifier as it contain '@' – special character

- Identifiers should **not** start with digits([**0-9**]) For example "123geeks" is a not a valid java identifier

- Java identifiers are **case-sensitive**

- There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only

- **Reserved Words** can't be used as an identifier For example "int while = 20;" is an invalid statement as while is a reserved word There are **53** reserved words in Java

Examples of valid identifiers:

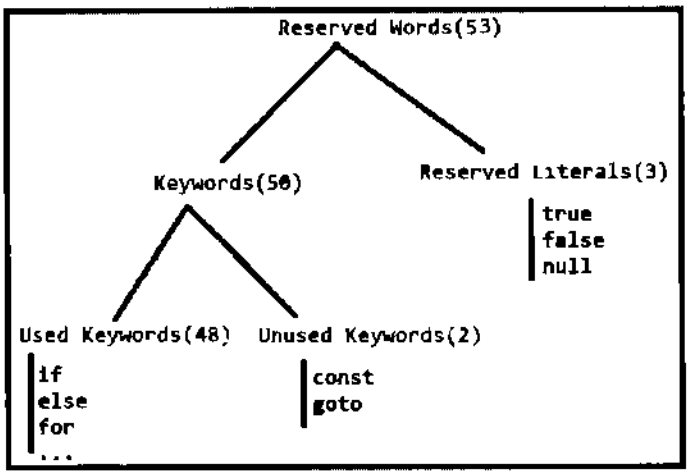
- MyVariable
- MYVARIABLE
- myvariable
- x
- i
- x1
- i1
- _myvariable
- $myvariable
- sum_of_array
- geeks123

Examples of invalid identifiers:

- My Variable // contains a space
- 123geeks // Begins with a digit
- a+c // plus sign is not an alphanumeric character
- variable-2 // hyphen is not an alphanumeric character
- sum_&_difference // ampersand is not an alphanumeric character

Reserved Words

Any programming language reserves some words to represent functionalities defined by that language. These words are called reserved words They can be briefly categories into two parts · **keywords**(50) and **literals**(3). Keywords define functionalities and literals define a value



Identifiers are used by symbol tables in various analyzing phases (like lexical, syntax, semantic) of compiler architecture

Data types

There are majorly two types of languages First one is **statically typed language** where each variable and expression type is already known at compile time Once a variable is declared to be of a certain data type, it cannot hold values of other data types Example C,C++, Java. Other, **Dynamically typed languages:** These languages can receive different data types over the time (i e) Ruby, Python

Java is **statically typed and also a strongly typed language** because in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types

Java has two categories of data

- Primitive data (e.g , number, character)
- Object data (programmer created types)

Primitive data

Primitive data are only single values; they have no special capabilities There are 8 primitive data types

Type	Description	Default	Size	Example Literals
boolean	true or false	false	1 bit	true, false
byte	twos complement integer	0	8 bits	(none)
char	Unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\'', '\n', 'ß'
short	twos complement integer	0	16 bits	(none)
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L
float	IEEE 754 floating point	0.0	32 bits	1 23e100f, -1.23e-100f, 3f, 3.14F
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d

Boolean:

Boolean data type represents only one bit of information **either true or false** Values of type Boolean are not converted implicitly or explicitly (with casts) to any other type But the programmer can easily write conversion code

```
// A Java program to demonstrate boolean data type
class Egboolean
{
   public static void main(String args[])
   {
      boolean b = true,
      if (b == true)
         System out println("Hi"),
   }
}
Output
Hi
```

Byte

The byte data type is an 8-bit signed two's complement integer The byte data type is useful for saving memory in large arrays

- Size: 8-bit
- Value -128 to 127

```
// Java program to demonstrate byte data type in Java
class Egbyte{
   public static void main(String args[])    {
      byte a = 126,
       // byte is 8 bit value
      System out.println(a),
      a++,
      System out println(a),
      // It overflows here because byte can hold values from -128 to 127
      a++,
      System out println(a),
// Looping back within the range
      a++,
      System out print(a),
   }
}

Output
126 127 -128 -127
```

Short

The short data type is a 16-bit signed two's complement integer Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters

- **Size:** 16 bit
- **Value:** -32,768 to 32,767 (inclusive)

Int

It is a 32-bit signed two's complement integer

- **Size:** 32 bit
- **Value:** -2^{31} to $2^{31}-1$

Note In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has value in range [0, $2^{32}-1$] Use the Integer class to use int data type as an unsigned integer

Long:

The long data type is a 64-bit two's complement integer.

- Size 64 bit
- Value -2^{63} to $2^{63}-1$

Note: In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$. The Long class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long

Floating point Numbers: float and double float

The float data type is a single-precision 32-bit IEEE 754 floating point Use a float (instead of double) if you need to save memory in large arrays of floating point numbers

- **Size:** 32 bits
- **Suffix :** F/f Example 9 8f

Double:

The double data type is a double-precision 64-bit IEEE 754 floating point For decimal values, this data type is generally the default choice

Note Both float and double data types were designed especially for scientific calculations, where approximation errors are acceptable If accuracy is the most prior concern then, it is recommended not to use these data types and use BigDecimal class instead.

Char

The char data type is a single 16-bit Unicode character A char is a single character

- Value '\u0000' (or 0) to '\uffff' 65535

```
// Java program to demonstrate primitive data types in Java
class Example {
   public static void main(String args[])    {
      // declaring character
       char a = 'G',
      // Integer data type is generally used for numeric values
       int i=89,
```

```
        // use byte and short if memory is a constraint
          byte b = 4;
         // this will give error as number is larger than byte range byte b1 = 7888888955,
           short s = 56,
         // this will give error as number is larger than short range short s1 = 87878787878,
         // by default fraction value is double in java
          double d = 4 355453532,
        // for float use 'f' as suffix
          float f = 4 7333434f,
          System out.println("char " + a),
          System out println("integer " + i);
          System.out println("byte " + b),
          System out println("short " + s),
          System out println("float " + f);
          System out println("double " + d),
      }
}
Output
char G
integer 89
byte: 4
short 56
float 4.7333436
double: 4.355453532
```

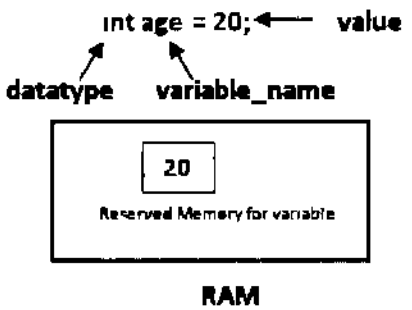
Variables:

A variable is the name given to a memory location It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location
- In Java, all the variables must be declared before they can be used

How to declare variables?

We can declare variables in java as follows:



datatype Type of data that can be stored in this variable
variable_name Name given to the variable
value. It is the initial value stored in the variable

Examples

```
float simpleInterest; //Declaring float variable
int time = 10, speed = 20, //Declaring and Initializing integer variable
char var = 'h', // Declaring and Initializing character variable
```

Types of variables

There are three types of variables in Java·

- Local Variables
- Instance Variables
- Static Variables

Local Variables.

A variable defined within a block or method or constructor is called local variable

- These variables are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared i e we can access these variable only within that block

```
public class StudentDetails{
  public void StudentAge()   {  //local variable age
    int age = 0;
    age = age + 5,
    System.out println("Student age is : " + age);
  }
  public static void main(String args[])   {
    StudentDetails obj = new StudentDetails(),
    obj StudentAge(),
  }
}
```

Output.

Student age is 5

In the above program the variable age is local variable to the function StudentAge() If we use the variable age outside StudentAge() function, the compiler will produce an error as shown in below program

```
public class StudentDetails{
  public void StudentAge()   {  //local variable age
    int age = 0,
    age = age + 5,
  }
```

```
    public static void main(String args[])
    {
      //using local variable age outside it's scope
      System out println("Student age is   " + age),
    }
}
```

Output

error cannot find symbol " + age),

Instance Variables

Instance variables are non-static variables and are declared in a class outside any method, constructor or block

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifiers then the default access specifier will be used

```
import java io *,
class Marks{   //These variables are instance variables. These variables are in a class and are
//not inside any function
   int engMarks,
   int mathsMarks,
   int phyMarks,
}
 class MarksDemo
{
   public static void main(String args[])
   {  //first object
      Marks obj1 = new Marks(),
      obj1.engMarks = 50,
      obj1 mathsMarks = 80,
      obj1 phyMarks = 90,
       //second object
      Marks obj2 = new Marks(),
      obj2 engMarks = 80,
      obj2 mathsMarks = 60,
      obj2 phyMarks = 85,
       //displaying marks for first object
      System out println("Marks for first object "),
      System out println(obj1 engMarks),
      System out println(obj1 mathsMarks),
```

```
        System out println(obj1 phyMarks),
        //displaying marks for second object
        System out println("Marks for second object "),
        System out println(obj2 engMarks),
        System out println(obj2 mathsMarks),
        System out println(obj2 phyMarks),
      }
   }
```

Output

```
Marks for first object
50
80
90
Marks for second object
80
60
85
```

As the above program the variables, *engMarks* , *mathsMarks* , *phyMarks*are instance variables In case we have multiple objects as in the above program, each object will have its own copies of instance variables It is clear from the above output that each object will have its own copy of instance variable

Static Variables

Static variables are also known as Class variables

- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create
- Static variables are created at start of program execution and destroyed automatically when execution ends

To access static variables, we need not to create any object of that class, we can simply access the variable as

class_name variable_name.

```
   import java io *,
   class Emp {
     // static variable salary
     public static double salary,
     public static String name = "Harsh",
   }
   public class EmpDemo{
```

```
    public static void main(String args[]) {
        //accessing static variable without object
     Emp salary = 1000;
     System out println(Emp name + "'s average salary " +
  Emp salary). }}
```

Output.
Harsh's average salary:1000.0

Review Questions

1. Define reserved words
2. Explain primitive and non – primitive data types
3. Describe types of variables with examples
4 Discuss the rules for framing identifiers.

CHAPTER 8

OPERATORS, CONDITIONAL STATEMENTS AND LOOPING

Operators:

Java provides many types of operators which can be used according to the need They are classified based on the functionality they provide. Some of the types are –

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4 Relational Operators
5 Logical Operators
6. Ternary Operator
7 Bitwise Operators
8. Shift Operators
9. instance of operator
10. Precedence and Associativity

Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.

- ***** : Multiplication
- **/** : Division
- **%** : Modulo
- **+** : Addition
- **–** : Subtraction

```
// Java program to illustrate arithmetic operators
public class operators
{
  public static void main(String[] args)
  {
    int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30,
    String x = "Thank", y = "You",

    // + and - operator
    System out println("a + b = "+(a + b)),
    System out.println("a - b = "+(a - b));

    // + operator if used with strings it concatenates the given strings
    System out println("x + y = "+x + y),
```

```
        // * and / operator
        System out println("a * b = "+(a * b)),
        System out println("a / b = "+(a / b)),

        // modulo operator gives remainder
        // on dividing first operand with second
        System.out println("a % b = "+(a % b)),

        // if denominator is 0 in division Then Arithmetic exception is thrown
        // uncommenting below line would throw an exception
        System.out.println(a/c),
    }
}
```

Output:

```
1  a+b = 30
2  a-b = 10
3  x+y = ThankYou
4  a*b = 200
5  a/b = 2
6  a%b = 0
7  error
```

Unary Operators: Unary operators needs only one operand They are used to increment, decrement or negate a value

- – :**Unary minus**, used for negating the values
- + :**Unary plus**, used for giving positive values Only used when deliberately converting a negative value to positive
- ++ :**Increment operator**, used for incrementing the value by 1 There are two varieties of increment operator
  - **Post-Increment :** Value is first used for computing the result and then incremented
  - **Pre-Increment :** Value is decremented first and then result is computed
- — : **Decrement operator**, used for incrementing the value by 1 There are two varieties of increment operator
  - **Post-decrement :** Value is first used for computing the result and then decremented
  - **Pre-Decrement :** Value is incremented first and then result is computed
- ! : **Logical not operator**, used for inverting a boolean value

```
// Java program to illustrate unary operators
public class operators
{
   public static void main(String[] args)
   {
      int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30,
      boolean condition = true;
       // pre-increment operator a = a+1 and then c = a,
      c = ++a,
      System.out println("Value of c (++a) = " + c),

      // post increment operator c=b then b=b+1
      c = b++,
      System out println("Value of c (b++) = " + c),

      // pre-decrement operator d=d-1 then c=d
      c = --d,
      System out println("Value of c (--d) = " + c),

      // post-decrement operator c=e then e=e-1
      c = --e,
      System out println("Value of c (--e) = " + c),

      // Logical not operator
      System.out println("Value of !condition =" + !condition),
   }
}
```

Output:

```
Value of c (++a) = 21
Value of c (b++) = 10
Value of c (--d) = 19
Value of c (--e) = 39
Value of !condition =false
```

Assignment Operator : '=' Assignment operator is used to assign a value to any variable It has a right to left associativity, i e value given on right hand side of operator is assigned to the variable on the left and therefore right hand side value must be declared before using it or should be a constant.General format of assignment operator is,

variable = value,

In many cases assignment operator can be combined with other operators to build a shorter version of statement called **Compound Statement**. For example, instead of a = a+5 , we can write a += 5

- +=, for adding left operand with right operand and then assigning it to variable on the left
- -=, for subtracting left operand with right operand and then assigning it to variable on the left
- *=, for multiplying left operand with right operand and then assigning it to variable on the left
- /=, for dividing left operand with right operand and then assigning it to variable on the left
- ^=, for raising power of left operand to right operand and assigning it to variable on the left
- %=, for assigning modulo of left operand with right operand and then assigning it to variable on the left

```
int a = 5;
a += 5; //a = a+5;
    // Java program to illustrate assignment operators
    public class operators
    {
       public static void main(String[] args)
       {
          int a = 20, b = 10, c, d, e = 10, f = 4, g = 9,
           // simple assignment operator
          c = b,
          System out println("Value of c = " + c),
           // This following statement would throw an exception as value of right operand must be
          //initialized before assignment, and the program would not compile  // c = d,
           // instead of below statements, shorthand assignment operators can be used to
          // provide same functionality
          a = a + 1,
          b = b - 1;
          e = e * 2,
          f = f / 2,
          System out println("a,b,e,f = " + a + ","
                        + b + "," + e + "," + f);
          a = a - 1,
          b = b + 1,
          e = e / 2,
```

```
        f = f * 2,

        // shorthand assignment operator
        a += 1,
        b -= 1,
        e *= 2,
        f /= 2;
        System.out.println("a,b,e,f (using shorthand operators)= " +
                        a + "," + b + "," + e + "," + f);
    }
}
```

Output :
Value of c =10
a,b,e,f = 21,9,20,2
a,b,e,f (using shorthand operators)= 21,9,20,2

Relational Operators : These operators are used to check for relations like equality, greater than, less than. They return boolean result after the comparison and are extensively used in looping statements as well as conditional if else statements General format is, variable **relation_operator** value.Some of the relational operators are-

- **==, Equal to :** returns true of left hand side is equal to right hand side
- **!=, Not Equal to :** returns true of left hand side is not equal to right hand side
- **<, less than :** returns true of left hand side is less than right hand side
- **<=, less than or equal to :** returns true of left hand side is less than or equal to right hand side
- **>, Greater than :** returns true of left hand side is greater than right hand side.
- **>=, Greater than or equal to:** returns true of left hand side is greater than or equal to right hand side

```
// Java program to illustrate relational operators
public class operators
{
    public static void main(String[] args)
    {
        int a = 20, b = 10,
        String x = "Thank", y = "Thank",
        int ar[] = { 1, 2, 3 },
        int br[] = { 1, 2, 3 };
```

```
        boolean condition = true,

        //various conditional operators
        System out println("a == b  " + (a == b)),
        System out println("a < b  " + (a < b)),
        System out println("a <= b  " + (a <= b)),
        System out println("a > b  " + (a > b)),
        System out println("a >= b  " + (a >= b));
        System out println("a != b  " + (a != b)),

        // Arrays cannot be compared with relational operators because objects
        // store references not the value
        System out.println("x == y   " + (ar == br)),
         System out println("condition==true  " + (condition == true)),
    }
}
```

Output :

a==b .false
a<b false
ab true
a>=b .true
a!=b true
x==y   false
condition==true true

Logical Operators: These operators are used to perform "logical AND" and "logical OR" operation, i e the function similar to AND gate and OR gate in digital electronics One thing to keep in mind is the second condition is not evaluated if the first one is false, i e it has short-circuiting effect Used extensively to test for several conditions for making a decision Conditional operators are-

- **&& , Logical AND :** returns true when both conditions are true
- || , **Logical OR :** returns true if at least one condition is true

```
// Java program to illustrate logical operators
public class operators
{
   public static void main(String[] args)
    {
```

```
        String x = "Sher",
        String y = "Locked";

        Scanner s = new Scanner(System in);
        System out.print("Enter username:"),
        String uuid = s next(),
        System.out.print("Enter password·");
        String upwd = s next();

        // Check if user-name and password match or not
        if ((uuid equals(x) && upwd equals(y)) ||
            (uuid.equals(y) && upwd equals(x))) {
          System.out println("Welcome user ");
        } else {
          System.out println("Wrong uid or password");
        }
      }
    }
```

Output :
Enter username:Sher
Enter password.Locked
Welcome user.

Ternary operator : Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary General format is-

condition ? if true : if false

The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

```
// Java program to illustrate max of three numbers using ternary operator.
public class operators
{
  public static void main(String[] args)
  {
    int a = 20, b = 10, c = 30, result;
     //result holds max of three numbers
    result = ((a > b) ? (a > c) ? a   c . (b > c) ? b : c),
    System out println("Max of three numbers = "+result);
  }
}
```

Output :

Max of three numbers = 30

Bitwise Operators : These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types They are used when performing update and query operations of Binary indexed tree.

- **& , Bitwise AND operator:** returns bit by bit AND of input values.
- **| , Bitwise OR operator:** returns bit by bit OR of input values
- **^ , Bitwise XOR operator:** returns bit by bit XOR of input values.
- **~ , Bitwise Complement Operator:** This is a unary operator which returns the one's compliment representation of the input value, i.e with all bits inversed

```
// Java program to illustrate bitwise operators
public class operators
{
  public static void main(String[] args)
  {
    int a = 0x0005,
    int b = 0x0007;
     // bitwise and 0101 & 0111=0101
    System out println("a&b = " + (a & b)),
     // bitwise and 0101 | 0111=0111
    System.out println("a|b = " + (a | b));
     // bitwise xor  0101 ^ 0111=0010
    System.out println("a^b = " + (a ^ b)),
     // bitwise and ~0101=1010
    System out.println("~a = " + ~a),
     // can also be combined with assignment operator to provide shorthand assignment
    // a=a&b
     a &= b;
    System out println("a= " + a),
  }
}
```

Output :

```
a&b = 5
a|b = 7
a^b = 2
~a = -6
a= 5
```

Shift Operators :These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two. General format-
number **shift_op** number_of_places_to_shift:

- **<< , Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.
- **>> , Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.
- **>>> , Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

```
// Java program to illustrate shift operators
public class operators
{
   public static void main(String[] args)
   {
      int a = 0x0005,
      int b = -10;
      // left shift operator  000 0101<<2 =0001 0100(20) similar to 5*(2^2)
      System out println("a<<2 = " + (a << 2));

      // right shift operator 0000 0101 >> 2 =0000 0001(1) similar to 5/(2^2)
      System.out.println("a>>2 = " + (a >> 2));

      // unsigned right shift operator
      System.out println("b>>>2 = "+ (b >>> 2)),
   }
}
```

Output :
a<>2 = 1
b>>>2 = 1073741821

instance of operator : Instance of operator is used for type checking It can be used to test if an object is an instance of a class, a subclass or an interface General format- object **instance of** class/subclass/interface

```
// Java program to illustrate instance of operator
class operators
{
   public static void main(String[] args)
   {

      Person obj1 = new Person();
      Person obj2 = new Boy();

      // As obj is of type person, it is not an instance of Boy or interface
      System.out.println("obj1 instanceof Person " + (obj1 instanceof Person));
      System.out.println("obj1 instanceof Boy: " + (obj1 instanceof Boy));
      System.out.println("obj1 instanceof MyInterface " + (obj1 instanceof MyInterface));

      // Since obj2 is of type boy, whose parent class is person and it implements the interface
      //MyInterface it is instance of all of these classes
      System.out.println("obj2 instanceof Person: " + (obj2 instanceof Person));
      System.out.println("obj2 instanceof Boy. " + (obj2 instanceof Boy));
      System.out.println("obj2 instanceof MyInterface " + (obj2 instanceof MyInterface));
   }
}

class Person
{
}
class Boy extends Person implements MyInterface
{
}
interface MyInterface
{
}
```

Output :

```
obj1 instanceof Person  true
obj1 instanceof Boy  false
obj1 instanceof MyInterface  false
obj2 instanceof Person  true
obj2 instanceof Boy: true
obj2 instanceof MyInterface true
```

Precedence and Associativity of Operators

Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator In such cases, these rules determine which part of equation to consider first as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude with top representing the highest precedence and bottom shows lowest precedence.

Operators	Associativity	Type
++ --	right to left	unary postfix
++ -- + - ! (*type*)	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
|	left to right	boolean logical inclusive OR
&&	left to right	conditional AND
||	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Precedence and Associativity: There is often confusion when it comes to hybrid equations that is equations having multiple operators The problem is which part to solve first There is a golden rule to follow in these situations If the operators have different precedence, solve the higher precedence first If they have same precedence, solve according to associativity, that is either from right to left or from left to right Explanation of below program is well written in comments within the program itself

```
public class operators
{
  public static void main(String[] args)
  {
    int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30,

    // precedence rules for arithmetic operators  (* = / = %) > (+ = -)
    // prints a+(b/d)
```

```
        System.out.println("a+b/d = "+(a + b / d)),
    // if same precendence then associative rules are followed  e/f -> b*d -> a+(b*d) -> a+(b*d)-(e/f)
        System out.println("a+b*d-e/f = "+(a + b * d - e / f)),
    }
}
```

Output:

```
a+b/d = 20
a+b*d-e/f = 219
```

Be a Compiler: Compiler in our systems uses lex tool to match the greatest match when generating tokens. This creates a bit of problem if overlooked For example, consider the statement **a=b+++c;**, to many of the readers this might seem to create compiler error. But this statement is absolutely correct as the token created by lex are a, =, b, ++, +, c Therefore this statement has similar effect of first assigning b+c to a and then incrementing b Similarly, **a=b+++++c;** would generate error as tokens generated are a, =, b, ++, ++, +, c. which is actually a error as there is no operand after second unary operand.

```
public class operators
{
  public static void main(String[] args)
  {
    int a = 20, b = 10, c = 0,
    // a=b+++c is compiled as b++ +c a=b+c then b=b+1
    a = b+++c;
    System.out.println("Value of a(b+c),b(b+1),c = " + a + "," + b + "," + c),
    // a=b+++++c is compiled as b++ ++ +c which gives error  a=b+++++c,
    // System.out.println(b+++++c),
  }
}
```

Output:

```
Value of a(b+c),b(b+1),c = 10,11,0
```

Using + over (): When using + operator inside system out println() make sure to do addition suing parenthesis. If we write something before doing addition, then string addition takes place, that is associativity of addition is left to right and hence integers are added to string first producing a string, and string objects concatenates when using +, therefore it can create unwanted results.

```
public class operators
{
  public static void main(String[] args)
  {
    int x = 5, y = 8,
     // concatenates x and y as first x is added to "concatenation (x+y) = "
     // producing "concatenation (x+y) = 5" and then 8 is further concatenated.
     System out println("Concatenation (x+y)= " + x + y),
      // addition of x and y
     System out println("Addition (x+y) = " + (x + y)),
  }
}
```

Output:

Concatenation (x+y)= 58
Addition (x+y) = 13

Conditional statements/Decision Making:

Decision Making in programming is similar to decision making in real life. In programming also we face some situations where we want a certain block of code to be executed when some condition is fulfilled A programming language uses control statements to control the flow of execution of program based on certain conditions These are used to cause the flow of execution to advance and branch based on changes to the state of a program

Statements:

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

These statements allow you to control the flow of your program's execution based upon conditions known only during run time

If- if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i e if a certain condition is true then a block of statement is executed otherwise not

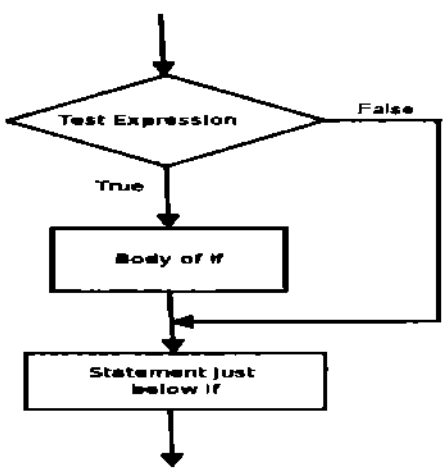
Syntax:

```
if(condition)
{
  // Statements to execute if condition is true
}
```

Here, **condition** after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements under it If we do not provide the curly braces '{' and '}' after **if(condition)** then by default if statement will consider the immediate one statement to be inside its block For example,

```
if(condition)
  statement1;
  statement2;
// Here if the condition is true, if block will consider only statement1 to be inside its block
```

Flowchart:



Example:

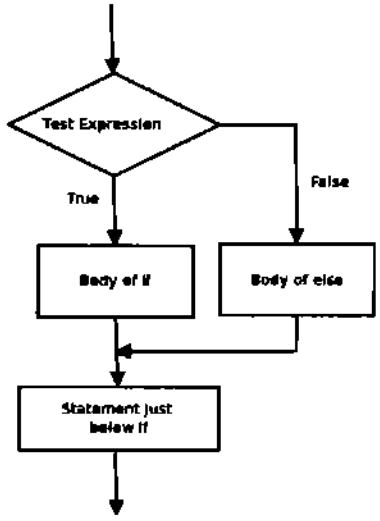
```
// Java program to illustrate If statement
class IfDemo {
   public static void main(String args[])    {
      int i = 10;
      if (i < 15)
         System.out.println("10 is less than 15");
 // This statement will be executed as if considers one statement by default
      System.out.println("I am Not in if"),
   }
}
```

Output: 10 is less than 15
I am Not in if

if-else The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't But what, if we want to do something else if the condition is false. Here comes the else statement We can use the else statement with if statement to execute a block of code when the condition is false

Syntax

```
if (condition)
{
  // Executes this block if
  // condition is true
}
else
{
  // Executes this block if
  // condition is false
}
```



Example

```
// Java program to illustrate if-else statement
class IfElseDemo {
   public static void main(String args[])   {
      int i = 10,
       if (i >15)
         System.out.println("i is smaller than 15"),
      else
         System.out.println("i is greater than 15"),
   }
}
```

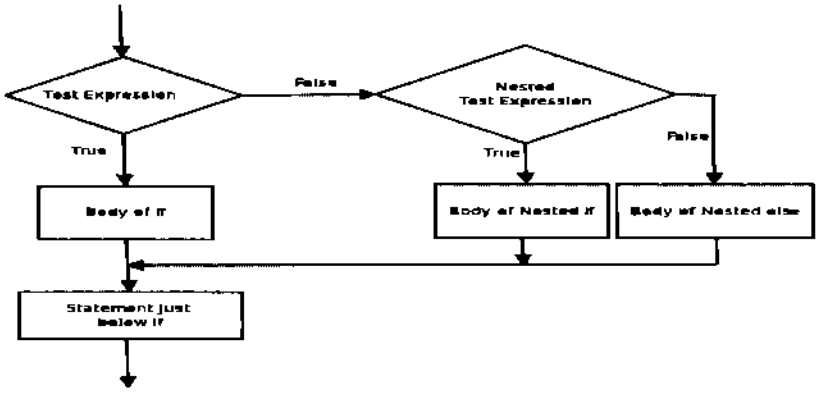
Output.

i is greater than 15

nested-if: A nested if is an if statement that is the target of another if or else Nested if statements means an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i e, we can place an if statement inside another if statement

Syntax

```
if (condition1)
{
  // Executes when condition1 is true
  if (condition2)
  {
    // Executes when condition2 is true
  }
}
```



Example:

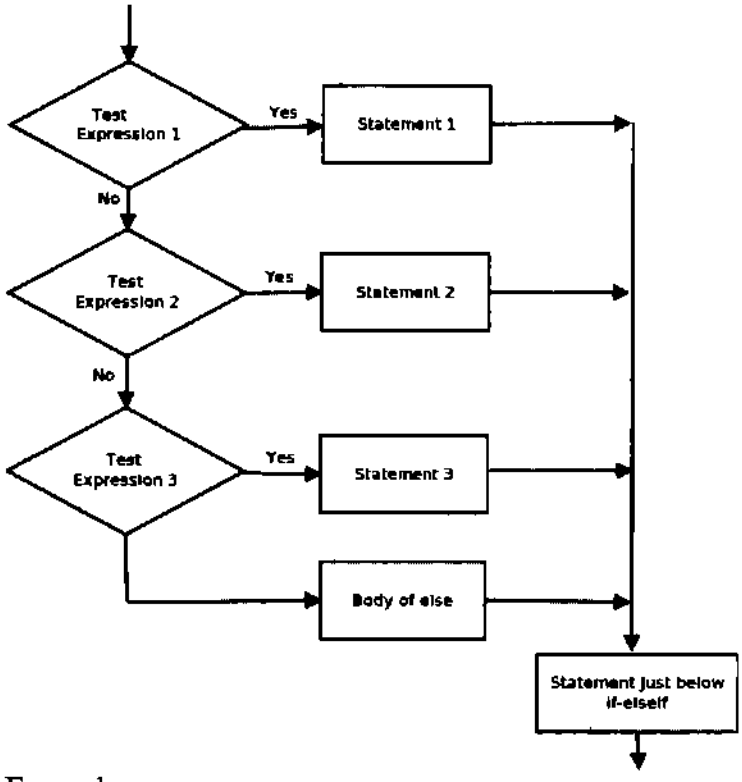
```
// Java program to illustrate nested-if statement
class NestedIfDemo{
   public static void main(String args[]) {
      int i = 10;
       if (i == 10)       {
         // First if statement
         if (i < 15)
            System out println("i is smaller than 15"),
          // Nested - if statement Will only be executed if statement above it is true
         if (i < 12)
            System out println("i is smaller than 12 too"),
         else
            System out println("i is greater than 15"),
      }   }}
```

Output

```
i is smaller than 15
i is smaller than 12 too
```

if-else-if ladder: Here, a user can decide among multiple options The if statements are executed from the top down As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed If none of the conditions is true, then the final else statement will be executed

```
if (condition)
   statement;
else if (condition)
   statement;
.
.
else
   statement,
```



Example:

```
// Java program to illustrate if-else-if ladder
class ifelseifDemo{
   public static void main(String args[])   {
      int i = 20;
      if (i == 10)
         System.out println("i is 10"),
      else if (i == 15)
         System.out.println("i is 15"),
      else if (i == 20)
         System out.println("i is 20");
```

```
        else
          System out.println("i is not present");
      }
    }
```

Output.

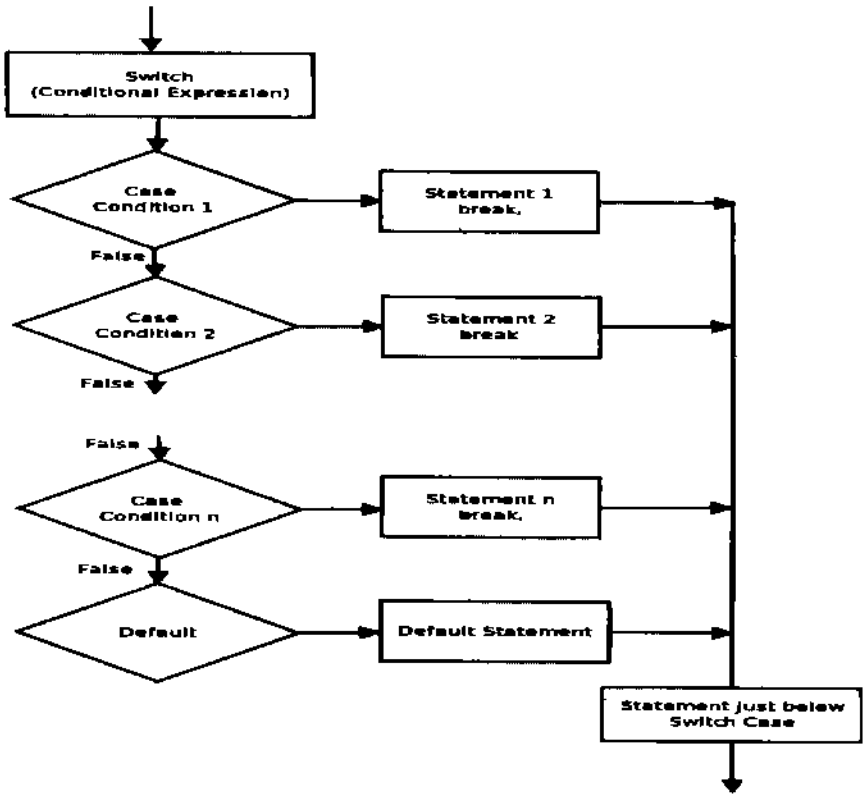
i is 20

switch-case· The switch statement is a multiway branch statement It provides an easy way to dispatch execution to different parts of code based on the value of the expression Syntax

```
switch (expression)
{
 case value1.
  statement1;
  break;
 case value2
  statement2,
  break,


 case valueN:
  statementN;
  break;
 default.
  statementDefault;
```

- Expression can be of type byte, short, int char or an enumeration. Beginning with JDK7, *expression* can also be of type String.
- Duplicate case values are not allowed
- The default statement is optional
- The break statement is used inside the switch to terminate a statement sequence
- The break statement is optional If omitted, execution will continue on into the next case



Example:

```
// Java program to illustrate switch-case
class SwitchCaseDemo{
   public static void main(String args[])    {
      int i = 9;
      switch (i)
      {
      case 0:
         System.out.println("i is zero.");
         break;
      case 1:
         System out println("i is one ");
         break;
      case 2:
         System.out.println("i is two.");
         break;
      default:
         System.out println("i is greater than 2.");
      }
   }
}
```

Output:

i is greater than 2.

jump: Java supports three jump statement **break, continue** and **return** These three statements transfer control to other part of the program

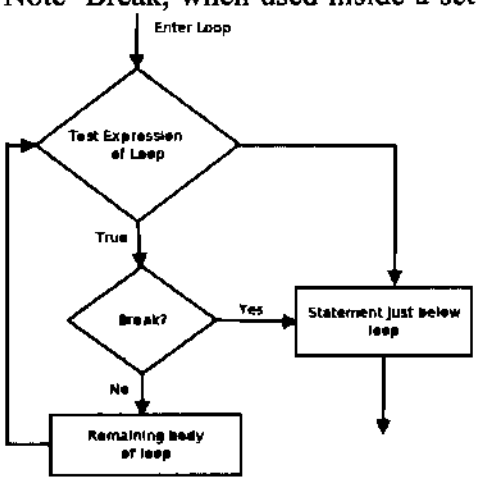
Break: In Java, break is majorly used for.

- Terminate a sequence in a switch statement (discussed above).
- To exit a loop.
- Used as a "civilized" form of goto

Using break to exit a Loop

Using break, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop

Note: Break, when used inside a set of nested loops, will only break out of the innermost loop



Example

```
// Java program to illustrate using break to exit a loop
class BreakLoopDemo {
  public static void main(String args[])  {
    // Initially loop is set to run from 0-9
    for (int i = 0; i < 10; i++)
    {
      // terminate loop when i is 5
      if (i == 5)
        break;
      System.out.println("i  " + i);
    }
    System.out.println("Loop complete ");
  }
}
```

Output

i: 0
i: 1
i: 2
i: 3
i 4
Loop complete.

Using break as a Form of Goto

Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner Java uses label A Label is use to identifies a block of code

Syntax

label
{
statement1,
statement2;
statement3,

}

Now, break statement can be use to jump out of target block. Note You cannot break to any label which is not defined for an enclosing block

Syntax

break label;

Example

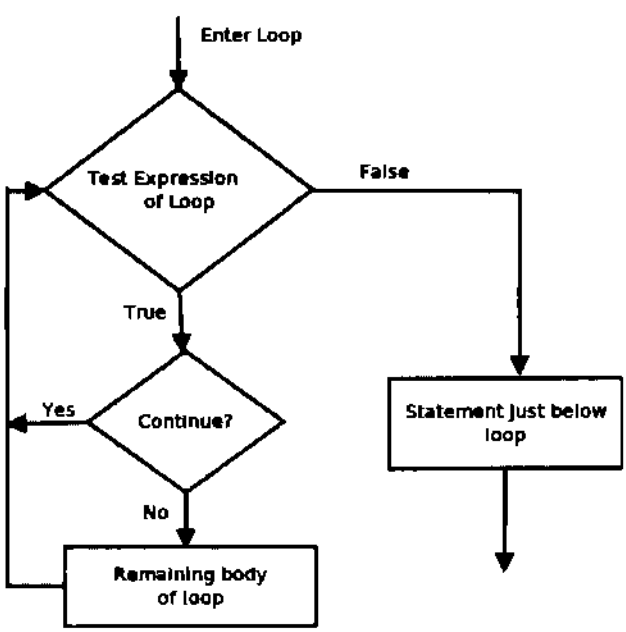
```
// Java program to illustrate using break with goto
class BreakLabelDemo
{
  public static void main(String args[])
  {
    boolean t = true,
    // label first
    first
    {
      // Illegal statement here as label second is not introduced yet break second,
      second
      {
        third
        {
          // Before break
          System out println("Before the break statement"),
```

```
            // break will take the control out of second label
            if (t)
               break second;
            System out.println("This won't execute ");
          }
          System.out println("This won't execute ");
        }
        // Third block
        System.out.println("This is after second block ");
      }
    }
}
```

Output:

Before the break.
This is after second block.

Continue: Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration This is, in effect, a goto just past the body of the loop, to the loop's end The continue statement performs such an action



Example:

// Java program to illustrate using continue in an if statement

```
class ContinueDemo
{
   public static void main(String args[])
   {
      for (int i = 0, i < 10; i++)
      {
         // If the number is even skip and continue
         if (i%2 == 0)
            continue,
          // If number is odd, print it
         System out print(i + " "),
      }
   }
}
```

Output

1 3 5 7 9

Return: The return statement is used to explicitly return from a method That is, it causes a program control to transfer back to the caller of the method

Example.

```
// Java program to illustrate using return
class Return{
   public static void main(String args[])   {
      boolean t = true,
      System out.println("Before the return "),
      if (t)
         return;
      // Compiler will bypass every statement after return
      System out println("This won't execute "),
   }
}
```

Output·

Before the return

Loops:

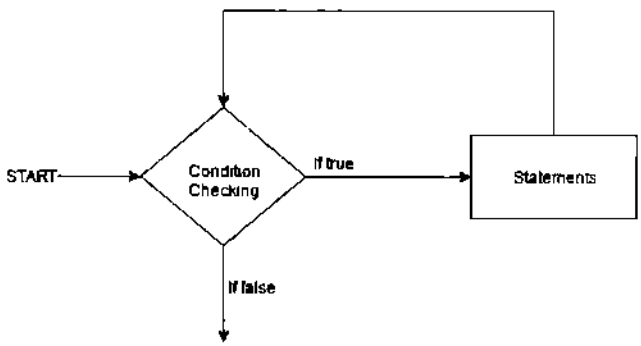
Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true Java provides three ways for executing the loops While all the ways provide similar basic functionality, they differ in their syntax and condition checking time

while loop: A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement

Syntax :

```
while (boolean condition)
{
  loop statements
}
```

Flowchart



- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed For this reason it is also called **Entry control loop**
- Once the condition is evaluated to true, the statements in the loop body are executed Normally the statements contain an update value for the variable being processed for the next iteration
- When the condition becomes false, the loop terminates which marks the end of its life cycle

```
// Java program to illustrate while loop
class whileLoopDemo{
   public static void main(String args[])   {
      int x = 1,
       // Exit when x becomes greater than 4
      while (x <= 4)      {
         System out println("Value of x " + x),
          //increment the value of x for next iteration
         x++;       }
   }}
```

Output:

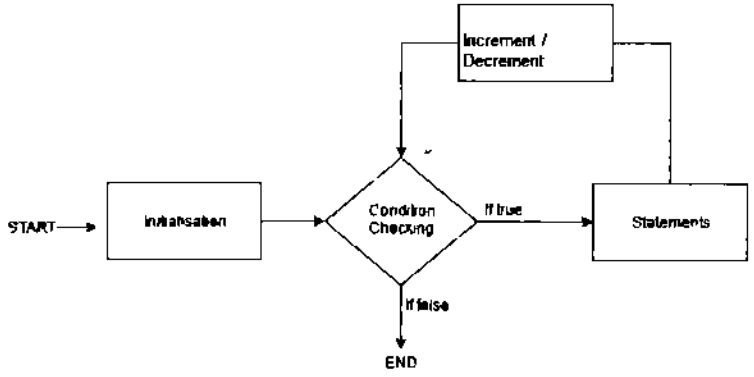
Value of x 1
Value of x 2
Value of x 3
Value of x 4

for loop: for loop provides a concise way of writing the loop structure Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping

Syntax:

```
for (initialization condition, testing condition,
                    increment/decrement)
{
   statement(s)
}
```

Flowchart



- **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only
- **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements
- **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed
- **Increment/ Decrement:** It is used for updating the variable for next iteration
- **Loop termination:** When the condition becomes false, the loop terminates marking the end of its life cycle

```
// Java program to illustrate for loop
class forLoopDemo{
   public static void main(String args[])   {
      // for loop begins when x=2 and runs till x <=4
      for (int x = 2, x <= 4, x++)
         System.out.println("Value of x " + x);   }}
```

Output:

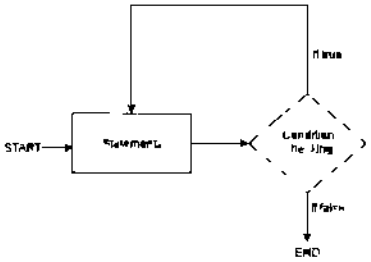
Value of x 2 Value of x 3 Value of x 4

do while: do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of **Exit Control Loop.**

Syntax:

```
do
{
  statements
}
while (condition),
```

Flowchart



- do while loop starts with the execution of the statement(s) There is no checking of any condition for the first time
- After the execution of the statements, and update of the variable value, the condition is checked for true or false value If it is evaluated to true, next iteration of loop starts
- When the condition becomes false, the loop terminates which marks the end of its life cycle
- It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop

```
// Java program to illustrate do-while loop
class dowhileloopDemo{
  public static void main(String args[])  {
    int x = 21,
    do      {
      //The line while be printer even if the condition is false
      System out println("Value of x " + x),
      x++,      }
    while (x < 20),   }}
```

Output:

Value of x 21

Review Questions

1 Define "+" operator in print statement
2 Explain operators and its precedence in detail
3 Describe types of conditional statements with examples
4 Discuss the concept of looping and its types

CHAPTER 9

INPUT AND OUTPUT STATEMENTS & EXCEPTION HANDLING

Using Buffered Reader Class

This is the Java classical method to take input, Introduced in JDK1 0 This method is used by wrapping the System in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line

Advantages

- The input is buffered for efficient reading

Drawback:

- The wrapping code is hard to remember

Program:

```
// Java program to demonstrate BufferedReader
import java.io BufferedReader,
    //Enter data using BufferReader
    BufferedReader reader = new BufferedReader(new InputStreamReader(System in)),
    // Reading data using readLine
    String name = reader readLine(),
     // Printing the read line
    System.out println(name),
```

Note To read other types, we use functions like Integer parseInt(), Double parseDouble() To read multiple values, we use split()

Using Scanner Class

This is probably the most preferred method to take input The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however it is also can be used to read input from the user in the command line

Advantages

- Convenient methods for parsing primitives (nextInt(), nextFloat(),) from the tokenized input
- Regular expressions can be used to find tokens

Drawback:

- The reading methods are not synchronized

```
// Java program to demonstrate BufferedReader
import java io BufferedReader,
    //Enter data using BufferReader
```

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System in)),
// Reading data using readLine
String name = reader.readLine().
// Printing the read line
System out.println(name),
```

Using Console Class

It has been becoming a preferred way for reading user's input from the command line In addition, it can be used for reading password-like input without echoing the characters entered by the user, the format string syntax can also be used (like System out print())

Advantages:

- Reading password without echoing the entered characters
- Reading methods are synchronized
- Format string syntax can be used

Drawback:

- Does not work in non-interactive environment

Exception:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled An exception can occur for many different reasons Following are some scenarios where an exception occurs

- A user has entered an invalid data
- A file that needs to be opened cannot be found
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- **Checked exceptions** – A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java io File,
import java io FileReader,
public class FilenotFound_Demo {
  public static void main(String args[]) {
    File file = new File("E //file txt"),
    FileReader fr = new FileReader(file),
  }
}
```

If you try to compile the above program, you will get the following exceptions

Output

```
C \>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error· unreported exception FileNotFoundException, must be caught
or declared to be thrown
    FileReader fr = new FileReader(file),
                 ^
1 error
```

Note – Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

- **Unchecked exceptions** – An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions** These include programming bugs, such as logic errors or improper use of an API Runtime exceptions are ignored at the time of compilation

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArravIndexOutOfBoundsExceptionexception* occurs

Example

```
public class Unchecked_Demo {
    public static void main(String args[]) {
    int num[] = {1, 2, 3, 4},
    System out println(num[5]),
  }
}
```

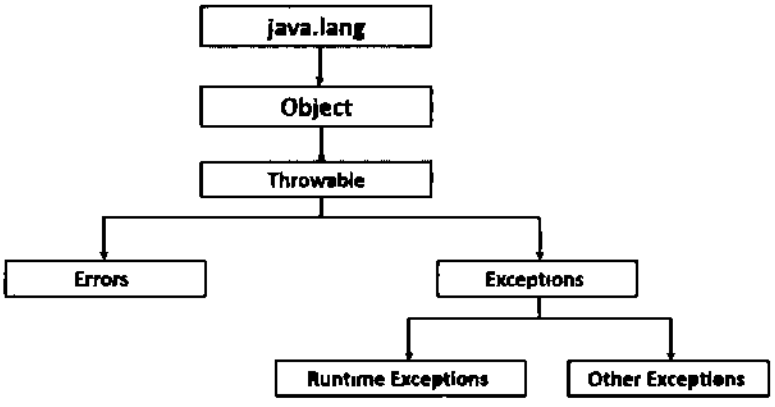
Output

```
Exception in thread "main" java lang ArrayIndexOutOfBoundsException
 5 at Exceptions Unchecked_Demo.main(Unchecked_Demo java)
```

- **Errors** – These are not exceptions at all, but problems that arise beyond the control of the user or the programmer Errors are typically ignored in your code because you can rarely do anything about an error For example, if a stack overflow occurs, an error will arise They are also ignored at the time of compilation

Exception Hierarchy

All exception classes are subtypes of the java lang.Exception class The exception class is a subclass of the Throwable class Other than the exception class there is another subclass called Error which is derived from the Throwable class Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs Errors are generated to indicate errors generated by the runtime environment. Example JVM is out of memory Normally, programs cannot recover from errors The Exception class has two main subclasses IOException class and RuntimeException Class



Following is a list of most common checked and unchecked Java's Built-in Exceptions

Exceptions Methods - Following is the list of important methods available in the Throwable class

Sr.No.	Method & Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred This message is initialized in the Throwable constructor
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage()
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System err, the error output stream
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords A try/catch block is placed around the code that might generate an exception Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following –

Syntax

```
try {
  // Protected code
}catch(ExceptionName e1) {
  // Catch block
}
```

The code which is prone to exceptions is placed in the try block When an exception occurs, that exception occurred is handled by catch block associated with it Every try block should be immediately followed either by a catch block or finally block

A catch statement involves declaring the type of exception you are trying to catch If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter

Example

The following is an array declared with 2 elements Then the code tries to access the 3rd element of the array which throws an exception

```
// File Name   ExcepTest java
import java io *,
public class ExcepTest {

  public static void main(String args[]) {
    try {
      int a[] = new int[2],
      System.out println("Access element three  " + a[3]),
    }catch(ArrayIndexOutOfBoundsException e) {
      System out.println("Exception thrown   " + e),
    }
    System out println("Out of the block"),
  }
}
```

Output

```
Exception thrown  :java lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks The syntax for multiple catch blocks looks like the following –

Syntax

```
try {
  // Protected code
}catch(ExceptionType1 e1) {
  // Catch block
}catch(ExceptionType2 e2) {
  // Catch block
}catch(ExceptionType3 e3) {
  // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there If not, the exception passes down to the second catch statement This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements

```
try {
  file = new FileInputStream(fileName),
  x = (byte) file read(),
}catch(IOException i) {
  i printStackTrace(),
  return -1,
}catch(FileNotFoundException f)// Not valid! {
  f printStackTrace(),
  return -1,
}
```

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code Here is how you would do it –

```
catch (IOException|FileNotFoundException ex) {
  logger log(ex),
  throw ex,
```

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword The throws keyword appears at the end of a method's signature You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly The following method declares that it throws a RemoteException –

Example

```
import java io.*,
public class className {
  public void deposit(double amount) throws RemoteException {
    // Method implementation
    throw new RemoteException(),   }
  // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas For example, the following method declares that it throws a RemoteException and an InsufficientFundsException –

Example

```
import java io *,
public class className {
  public void withdraw(double amount) throws RemoteException,
    InsufficientFundsException {
    // Method implementation
  }
  // Remainder of class definition
}
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code A finally block appears at the end of the catch blocks and has the following syntax –

Syntax

```
try {
  // Protected code
}catch(ExceptionType1 e1) {
  // Catch block
}catch(ExceptionType2 e2) {
  // Catch block
```

```
}catch(ExceptionType3 e3) {
  // Catch block
}finally {
  // The finally block always executes
}
```

Example

```
public class ExcepTest {
  public static void main(String args[]) {
    int a[] = new int[2];
    try {
      System.out.println("Access element three " + a[3]);
    }catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  " + e);
    }finally {
      a[0] = 6;
      System.out.println("First element value " + a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following −

- A catch clause cannot exist without a try statement
- It is not compulsory to have finally clauses whenever a try/catch block is present
- The try block cannot be present without either catch clause or finally clause
- Any code cannot be present in between the try, catch, finally blocks

The try-with-resources

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using **FileReader** and we are closing it using finally block.

Example

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {
  public static void main(String args[]) {
    FileReader fr = null;
    try {
```

```
    File file = new File("file txt"),
    fr = new FileReader(file), char [] a = new char[50],
    fr read(a),  // reads the content to the array
    for(char c  a)
    System out print(c),  // prints the characters one by one
  }catch(IOException e) {
    e printStackTrace(),
  }finally {
    try {
      fr.close(),
    }catch(IOException ex) {
      ex printStackTrace(),      }
  }  }}
```

try-with-resources, also referred as **automatic resource management**, is a new exception-handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block. To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.

Syntax

```
try(FileReader fr = new FileReader("file path")) {
  // use the resource
  }catch() {
    // body of catch
  } }
```

Following is the program that reads the data in a file using try-with-resources statement.

```
import java io FileReader,
import java io IOException,
public class Try_withDemo {
  public static void main(String args[]) {
    try(FileReader fr = new FileReader("E //file.txt")) {
      char [] a = new char[50],
      fr read(a),  // reads the contentto the array
      for(char c  a)
      System out print(c),  // prints the characters one by one
    }
catch(IOException e) {
      e printStackTrace(),
    }
  }}
```

Final:

Final keyword can be used along with variables, methods and classes.

- final variable
- final method
- final class

1) final variable

Final variables are nothing but constants We cannot change the value of a final variable once it is initialized Lets have a look at the below code

```
class Demo{
  final int MAX_VALUE=99,
  void myMethod(){
    MAX_VALUE=101,
  }
  public static void main(String args[]){
    Demo obj=new  Demo(),
    obj myMethod(),
  } }
```

Output:

```
Exception in thread "main" java lang Error  Unresolved compilation problem.
        The final field Demo MAX_VALUE cannot be assigned
        Demo myMethod(Details.java·6)   Demo main(Details java.10)
```

We got a compilation error in the above program because we tried to change the value of a final variable "MAX_VALUE" Note. It is considered as a good practice to have constant names in UPPER CASE(CAPS)

Blank final variable

A final variable that is not initialized at the time of declaration is known as **blank final variable** We **must initialize the blank final variable in constructor** of the class otherwise it will throw a compilation error (Error variable MAX_VALUE might not have been initialized) This is how a blank final variable is used in a class

```
class Demo{
  //Blank final variable
  final int MAX_VALUE,
  Demo(){
    //It must be initialized in constructor
    MAX_VALUE=100,
  }
  void myMethod(){
    System out println(MAX_VALUE),
  }
  public static void main(String args[]){
    Demo obj=new  Demo(),
    obj myMethod(),
  } }
```

Output: 100

What is the use of blank final variable?

Let's say we have a Student class which is having a field called Roll No Since Roll No should not be changed once the student is registered, we can declare it as a final variable in a class but we cannot initialize roll no in advance for all the students (otherwise all students would be having same roll no). In such case we can declare roll no variable as blank final and we initialize this value during object creation like this.

```
class StudentData{
  //Blank final variable
  final int ROLL_NO,
  StudentData(int rnum){
    //It must be initialized in constructor
    ROLL_NO=rnum,
  }
  void myMethod(){
    System out println("Roll no is "+ROLL_NO),
  }
  public static void main(String args[]){
    StudentData obj=new  StudentData(1234),
    obj.myMethod(),
  }
}
```

Output:

Roll no is 1234

2) Final method

A final method cannot be overridden Which means even though a sub class can be called by the final method of parent class without any issues it cannot override it

Example

```
class XYZ{
  final void demo(){
    System out.println("XYZ Class Method"),
  }
}
class ABC extends XYZ{
  void demo(){
    System out println("ABC Class Method"),
  }
  public static void main(String args[]){
    ABC obj= new ABC(),
    obj.demo(),
  }
}
```

The above program would throw a compilation error, however we can use the parent class final method in sub class without any issues. Let's have a look at this code This program would run fine as we are not overriding the final method That shows that final methods are inherited but they are not eligible for overriding

```
class XYZ{
  final void demo(){
    System out println("XYZ Class Method"),
  }
}
class ABC extends XYZ{
  public static void main(String args[]){
    ABC obj= new ABC(),
    obj demo(),
  }
}
```

Output:

XYZ Class Method

3) Final class

We cannot extend a final class (i e) once a class is declared with final keyword it cannot be inherited by any sub class Consider the below example

```
final class XYZ{
}
class ABC extends XYZ{
  void demo(){
    System.out println("My Method"),
  }
  public static void main(String args[]){
    ABC obj= new ABC(),
    obj demo(),
  }
}
```

Output:

The type ABC cannot subclass the final class XYZ

1) A constructor cannot be declared as final
2) Local final variable must be initializing during declaration
3) All variables declared in an interface are by default final
4) We cannot change the value of a final variable
5) A final method cannot be overridden
6) A final class not be inherited
7) If method parameters are declared final then the value of these parameters cannot be changed
8) final, finally and finalize are three different terms. Finally is used in exception handling and finalize is a method that is called by JVM during garbage collection

Finalize:

Finalize method in java is a special method much like the main method in java. finalize() is called before Garbage collector reclaim the Object, its last chance for any object to perform cleanup activity i e releasing any system resources held, closing connection if open etc The main issue with finalize method in Java is it's not guaranteed by JLS that it will be called by Garbage collector or exactly when it will be called, for example, an object may wait indefinitely after becoming eligible for garbage collection and before its finalize() method gets called Similarly even after finalize gets called it's not guaranteed it will be immediately collected Because of above reason it makes no sense to finalize method for releasing critical resources or perform any time critical activity inside finalize It may work in development in one JVM but may not work in other JVM

What is finalize method in Java

1) finalize() method is defined in java lang Object class, which means it available to all the classes for the sake of overriding finalize method is defined as protected

2) finalize method is not automatically chained like constructors If you are overriding finalize method then it's your responsibility to call finalize() method of the superclass, if you forgot to call then finalize of super class will never be called so it becomes critical to remember this and provide an opportunity to finalize of super class to perform cleanup The best way to call superclass finalize method is to call them in the finally block as shown in below example. This will guarantee that finalize of the parent class will be called in all condition except when JVM exits

```
@Override
  protected void finalize() throws Throwable {
     try{
        System out println("Finalize of Sub Class"),
        //release resources, perform cleanup ,
     }catch(Throwable t){
        throw t,
     }finally{
        System.out println("Calling finalize of Super Class"),
        super finalize(),
     }
}
```

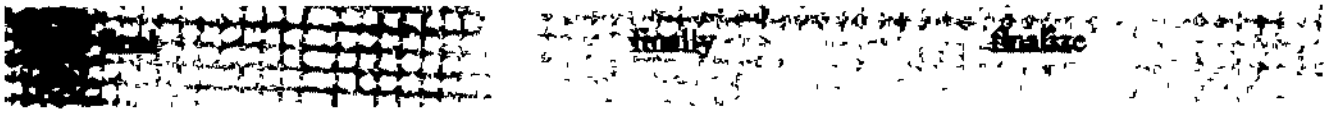
3) finalize method is called by garbage collection thread before collecting object and if not intended to be called like a normal method

4) finalize gets called only once by garbage collection thread if object revives itself from finalize method than finalize will not be called again

5) Any Exception is thrown by finalize method is ignored by garbage collection thread

6) There is one way to increase the probability of running of finalize method by calling System runFinalization() and Runtime getRuntime() runFinalization() These methods put more effort that JVM call finalize() method of all object which are eligible for garbage collection and whose finalize has not yet called It's not guaranteed, but JVM tries its best

Difference between final, finally and finalize



1)	Final is used to apply restrictions on class, method and variable Final class can't be inherited, final method can't be overridden and final variable value can't be changed	Finally is used to place important code, it will be executed whether exception is handled or not	Finalize is used to perform clean up processing just before object is garbage collected
2)	Final is a keyword.	Finally is a block	Finalize is a method

Review Questions

1 State the difference between println and print
2 Explain about Scanner class in detail
3 Describe exception handling with examples
4 Discuss the concept of **finally**

CHAPTER 10

ABSTRACT CLASS AND INTERFACE

ABSTRACT CLASS

Abstract classes are classes that contain one or more abstract methods An abstract method is a method that is declared, but contains no implementation Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user In other words, the user will have the information on what the object does instead of how it does it.
In Java, abstraction is achieved using Abstract classes and interfaces

- A class which contains the **abstract** keyword in its declaration is known as abstract class
- Abstract classes may or may not contain abstract methods, i.e., methods without body (public void get(),)
- But, if a class has at least one abstract method, then the class **must** be declared abstract
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it

Example

To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration

```
/* File name   Employee java */
public abstract class Employee {
  private String name,
  private String address,
  private int number,
  public Employee(String name, String address, int number) {
    System out println("Constructing an Employee"),
    this name = name,
    this address = address,
    this number = number,
  }
  public double computePay() {
   System out println("Inside Employee computePay"),
   return 0 0,   }
  public void mailCheck() {
    System out println("Mailing a check to " + this name + " " + this address),
  }
```

```
  public String toString() {
    return name + " " + address + " " + number;  }
  public String getName() {
    return name,  }
  public String getAddress() {
    return address,  }
  public void setAddress(String newAddress) {
    address = newAddress,  }
  public int getNumber() {
    return number;  }
}
```

You can observe that except abstract methods the Employee class is same as normal class in Java The class is now abstract, but it still has three fields, seven methods, and one constructor Now you can try to instantiate the Employee class in the following way −

```
/* File name  AbstractDemo java */
public class AbstractDemo {
  public static void main(String [] args) {
    /* Following is not allowed and would raise error */
    Employee e = new Employee("George W ", "Houston, TX", 43);
    System out println("\n Call mailCheck using Employee reference--"),
    e mailCheck();
  }
}
```

When you compile the above class, it gives you the following error − Employee java 46 Employee is abstract, cannot be instantiated Employee e = new Employee("George W ", "Houston, TX", 43),1 error

Inheriting the Abstract Class

We can inherit the properties of Employee class just like concrete class in the following way −

Example

```
/* File name  Salary java */
public class Salary extends Employee {
  private double salary,  // Annual salary
  public Salary(String name, Sting address, int number, double salary) {
    super(name, address, number),
    setSalary(salary),  }
  public void mailCheck() {
    System.out println("Within mailCheck of Salary class "),
    System out println("Mailing check to " + getName() + " with salary " + salary),
  }
```

```
public double getSalary() {
  return salary,
}
public void setSalary(double newSalary) {
  if(newSalary >= 0 0) {
    salary = newSalary,
  } }
public double computePay() {
  System out println("Computing salary pay for " + getName()),
  return salary/52,
} }
```

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below

```
/* File name  AbstractDemo java */
public class AbstractDemo {
  public static void main(String [] args) {
    Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600 00),
    Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00),
    System out println("Call mailCheck using Salary reference --"),
    s.mailCheck(),
    System out println("\n Call mailCheck using Employee reference--"),
    e mailCheck(); }}
```

Output

```
Call mailCheck using Salary reference -- Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600 0
 Call mailCheck using Employee reference-- Within mailCheck of Salary class
Mailing check to John Adams with salary 2400 0
```

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract

- **abstract** keyword is used to declare the method as abstract
- You have to place the **abstract** keyword before the method name in the method declaration
- An abstract method contains a method signature, but no method body
- Instead of curly braces, an abstract method will have a semoi colon (,) at the end

Following is an example of the abstract method

Example

```
public abstract class Employee {
  private String name,
  private String address,
  private int number,
   public abstract double computePay(),
  // Remainder of class definition}
```

Declaring a method as abstract has two consequences –

- The class containing it must be declared as abstract
- Any class inheriting the current class must either override the abstract method or declare itself as abstract

Suppose Salary class inherits the Employee class, then it should implement the **computePay()** method as shown below –

```
/* File name  Salary java */
public class Salary extends Employee {
  private double salary,  // Annual salary
  public double computePay() {
    System out println("Computing salary pay for " + getName());
    return salary/52,  }
  // Remainder of class definition}
```

Interface:

An interface is a reference type in Java It is similar to class It is a collection of abstract methods A class implements an interface, thereby inheriting the abstract methods of the interface Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types Method bodies exist only for default methods and static methods Writing an interface is similar to writing a class But a class describes the attributes and behaviors of an object And an interface contains behaviors that a class implements Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class

An interface is similar to a class in the following ways –

- An interface can contain any number of methods
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file
- The byte code of an interface appears in a **.class** file
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name

However, an interface is different from a class in several ways, including –

- You cannot instantiate an interface
- An interface does not contain any constructors
- All of the methods in an interface are abstract
- An interface cannot contain instance fields The only fields that can appear in an interface must be declared both static and final
- An interface is not extended by a class, it is implemented by a class
- An interface can extend multiple interfaces

Declaring Interfaces

The **interface** keyword is used to declare an interface Here is a simple example to declare an interface –

Example

```
/* File name  NameOfInterface java */
import java lang *,
// Any number of import statements
public interface NameOfInterface {
  // Any number of final, static fields
  // Any number of abstract method declarations
}
```

Interfaces have the following properties –

- An interface is implicitly abstract You do not need to use the **abstract** keyword while declaring an interface
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed
- Methods in an interface are implicitly public

Example

```
/* File name  Animal java */
interface Animal {
  public void eat(),
  public void travel(),
}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface If a class does not perform all the behaviors of the interface, the class must declare itself as abstract A class uses the **implements** keyword to implement an interface The implements keyword appears in the class declaration following the extends portion of the declaration

Example

```
/* File name  MammalInt.java */
public class MammalInt implements Animal {
  public void eat() {
    System out println("Mammal eats"),
  }
  public void travel() {
    System out println("Mammal travels"),
  }
  public int noOfLegs() {
    return 0;
  }
  public static void main(String args[]) {
    MammalInt m = new MammalInt(),
    m eat(),
    m travel(),
  }
}
```

Output

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces, there are several rules to be followed –

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods
- An implementation class itself can be abstract and if so, interface methods need not be implemented

When implementation interfaces, there are several rules –

- A class can implement more than one interface at a time
- A class can extend only one class, but implement many interfaces
- An interface can extend another interface, in a similar way as a class can extend another class

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface The following Sports interface is extended by Hockey and Football interfaces.

Example

```
// Filename  Sports java
public interface Sports {
  public void setHomeTeam(String name),
  public void setVisitingTeam(String name),
}
// Filename· Football java
public interface Football extends Sports {
  public void homeTeamScored(int points),
  public void visitingTeamScored(int points),
  public void endOfQuarter(int quarter),
}
// Filename  Hockey java
public interface Hockey extends Sports {
  public void homeGoalScored();
  public void visitingGoalScored(),
  public void endOfPeriod(int period),
  public void overtimePeriod(int ot),
}
```

The Hockey interface has four methods, but it inherits two from Sports, thus, a class that implements Hockey needs to implement all six methods Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports

Extending Multiple Interfaces

A Java class can only extend one parent class Multiple inheritance is not allowed Interfaces are not classes, however, and an interface can extend more than one parent interface The extends keyword is used once, and the parent interfaces are declared in a comma-separated list For example, if the Hockey interface extended both Sports and Event, it would be declared as –

Example: public interface Hockey extends Sports, Event

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods Abstract class and interface both can't be instantiated But there are many differences between abstract class and interface that are given below

Abstract class	**Interface**
1) Abstract class can **have abstract and non-abstract** methods	Interface can have **only abstract** methods Since Java 8, it can have **default and static methods** also
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables**	Interface has **only static and final variables**
4) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class**
5) The **abstract keyword** is used to declare abstract class	The **interface keyword** is used to declare interface

6) **Example:**
public abstract class Shape
{
public abstract void draw();
}
	Example:
	public interface Drawable
	{
	void draw();
	}

Review Questions

1 State the difference between abstract and interface
2 Explain about multiple interface
3 Describe abstract class and method with examples
4 Discuss the concept of interface inheritance

CHAPTER 11

THREAD AND MULTITHREAD

Thread :

A thread, in the context of Java, is the path followed when executing a program All Java programs have at least one thread, known as the main thread, which is created by the JVM at the program's start, when the main() method is invoked with the main thread In Java, creating a thread is accomplished by implementing an interface and extending a class Every Java thread is created and controlled by the java lang Thread class When a thread is created, it is assigned a priority The thread with higher priority is executed first, followed by lower-priority threads The JVM stops executing threads under either of the following conditions

- If the exit method has been invoked and authorized by the security manager
- All the daemon threads of the program have died

How to create thread: There are two ways to create a thread:

1 By extending Thread class
2 By implementing Runnable interface

Thread class

Thread class provide constructors and methods to create and perform operations on a thread Thread class extends Object class and implements Runnable interface

Commonly used Constructors of Thread class

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class

public void run(): is used to perform action for a thread
public void start(): starts the execution of the thread JVM calls the run() method on the thread
public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds
public void join(): waits for a thread to die
public void join(long miliseconds): waits for a thread to die for the specified miliseconds
public int getPriority(): returns the priority of the thread
public int setPriority(int priority): changes the priority of the thread
public String getName(): returns the name of the thread
public void setName(String name): changes the name of the thread
public Thread currentThread(): returns the reference of currently executing thread
public int getId(): returns the id of the thread
public Thread.State getState(): returns the state of the thread

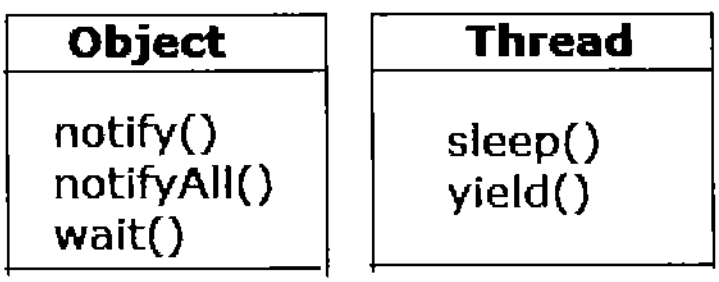
public boolean isAlive(): tests if the thread is alive
public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute
public void suspend(): is used to suspend the thread(depricated)
public void resume(): is used to resume the suspended thread(depricated)
public void stop(): is used to stop the thread(depricated)
public boolean isDaemon(): tests if the thread is a daemon thread
public void setDaemon(boolean b): marks the thread as daemon or user thread
public void interrupt(): interrupts the thread
public boolean isInterrupted(): tests if the thread has been interrupted
public static boolean interrupted(): tests if the current thread has been interrupted

MULTITHREADING:

Multithreading refers to two or more tasks executing concurrently within a single program A thread is an independent path of execution within a program Many threads can run concurrently within a program Every thread in Java is created and controlled by the **java.lang.Thread class** A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously Multithreading has several advantages over Multiprocessing such as,

- Threads are lightweight compared to processes
- Threads share the same address space and therefore can share both data and code
- Context switching between threads is usually less expensive than between processes
- Cost of thread intercommunication is relatively low that that of process intercommunication
- Threads allow different tasks to be performed concurrently

The following figure shows the methods that are members of the Object and Thread Class



THREAD CREATION

There are two ways to create thread in java,

- Implement the Runnable interface (java lang Runnable)
- By Extending the Thread class (java lang Thread)

IMPLEMENTING THE RUNNABLE INTERFACE

The Runnable Interface Signature

public interface Runnable {

void run(),

One way to create a thread in java is to implement the Runnable Interface and then instantiate an object of the class We need to override the run() method into our class which is the only method that needs to be implemented The run() method contains the logic of the thread

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

1. A class implements the Runnable interface, providing the run() method that will be executed by the thread An object of this class is a Runnable object

2 An object of Thread class is created by passing a Runnable object as argument to the Thread constructor The Thread object now has a Runnable object that implements the run() method

3 The start() method is invoked on the Thread object created in the previous step The start() method returns immediately after a thread has been spawned

4 The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception

Below is a program that illustrates instantiation and running of threads using the runnable interface instead of extending the Thread class To start the thread you need to invoke the **start()** method on your object.

```
class RunnableThread implements Runnable {
        Thread runner,
        public RunnableThread() {
        }
        public RunnableThread(String threadName) {
                runner = new Thread(this, threadName), // (1) Create a new thread
                System out println(runner getName()),
                runner start(), // (2) Start the thread
        }
        public void run() {
                //Display info about this particular thread
                System out println(Thread currentThread()),
        }}
public class RunnableExample {
        public static void main(String[] args) {
                Thread thread1 = new Thread(new RunnableThread(), "thread1"),
                Thread thread2 = new Thread(new RunnableThread(), "thread2"),
                RunnableThread thread3 = new RunnableThread("thread3"),
                //Start the threads
                thread1 start(),
                thread2 start(),
                try {
```

```
                //delay for one second
                Thread currentThread() sleep(1000),
            } catch (InterruptedException e) {
            }
            //Display info about the main thread
            System out println(Thread currentThread()),
    }}
```

Output

thread3
Thread[thread1,5,main] Thread[thread2,5,main] Thread[thread3,5,main] Thread[main,5,main]
private

This approach of creating a thread by implementing the Runnable Interface must be used whenever the class being used to instantiate the thread object is required to extend some other class -

EXTENDING THREAD CLASS

The procedure for creating threads based on extending the Thread is as follows

1 A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread

2 This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call

3 The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running

Below is a program that illustrates instantiation and running of threads by extending the Thread class instead of implementing the Runnable interface To start the thread you need to invoke the **start()**method on your object

```
class XThread extends Thread {
    XThread() {
    }
    XThread(String threadName) {
        super(threadName), // Initialize thread
        System out println(this),
        start(),
    }
    public void run() {
        //Display info about this particular thread
        System out println(Thread currentThread() getName()),
    }}
public class ThreadExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new XThread(), "thread1"),
        Thread thread2 = new Thread(new XThread(), "thread2"),
        //          The below 2 threads are assigned default names
        Thread thread3 = new XThread(),
```

```
        Thread thread4 = new XThread();
        Thread thread5 = new XThread("thread5");
        //Start the threads
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
        try {
//The sleep() method is invoked on the main thread to cause a one second delay
            Thread.currentThread().sleep(1000);
        } catch (InterruptedException e) {
        }
        //Display info about the main thread
        System.out.println(Thread.currentThread());
    }
}
```

Output

```
Thread[thread5,5,main] thread1
thread5
thread2
Thread-3
Thread-2
Thread[main,5,main]
```

When creating threads, there are two reasons why implementing the Runnable interface may be preferable to extending the Thread class

- Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive

THREAD SYNCHRONIZATION

With respect to multithreading, Synchronization is a process of controlling the access of shared resources by the multiple threads in such a manner that only one thread can access a particular resource at a time In non synchronized multithreaded application, it is possible for one thread to modify a shared object while another thread is in the process of using or updating the object's value Synchronization prevents such type of data corruption which may otherwise lead to dirty reads and significant errors Generally critical sections of the code are usually marked with synchronized keyword Examples of using Thread Synchronization is in "The Producer/Consumer Model"

Locks are used to synchronize access to a shared resource A lock can be associated with a shared resource Threads gain access to a shared resource by first acquiring the lock associated with the object/block of code At any given time, at most only one thread can hold the lock and thereby have access to the shared resource A lock thus implements mutual exclusion

The object lock mechanism enforces the following rules of synchronization

A thread must acquire the object lock associated with a shared resource, before it can enter the shared resource The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with the shared resource If a thread cannot immediately acquire the object lock, it is blocked, that is, it must wait for the lock to become available When a thread exits a shared resource, the runtime system ensures that the object lock is also relinquished If another thread is waiting for this object lock, it can proceed to acquire the lock in order to gain access to the shared resource

Classes also have a class-specific lock that is analogous to the object lock Such a lock is actually a lock on the java lang Class object associated with the class Given a class A, the reference A class denotes this unique Class object The class lock can be used in much the same way as an object lock to implement mutual exclusion There can be 2 ways through which synchronized can be implemented in Java

- synchronized methods
- synchronized blocks

Synchronized statements are same as synchronized methods A synchronized statement can only be executed after a thread has acquired the lock on the object/class referenced in the synchronized statement

SYNCHRONIZED METHODS

Synchronized methods are methods that are used to control access to an object A thread only executes a synchronized method after it has acquired the lock for the method's object or class If the lock is already held by another thread, the calling thread waits A thread relinquishes the lock simply by returning from the synchronized method, allowing the next thread waiting for this lock to proceed Synchronized methods are useful in situations where methods can manipulate the state of an object in ways that can corrupt the state if executed concurrently This is called a race condition It occurs when two or more threads simultaneously update the same value, and as a consequence, leave the value in an undefined or inconsistent state While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait until it gets the lock This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object Such a method can invoke other synchronized methods of the object without being blocked The non-synchronized methods of the object can of course be called at any time by any thread Below is an example shows how synchronized methods and object locks are used to coordinate access to a common object by multiple threads If the 'synchronized' keyword is removed, the message is displayed in random fashion

```
public class SyncMethodsExample extends Thread {
    static String[] msg = { "Beginner", "java", "tutorial,", " ,", "com", "is", "the", "best" },
    public SyncMethodsExample(String id) {
        super(id),     }
    public static void main(String[] args) {
```

```
        SyncMethodsExample thread1 = new SyncMethodsExample("thread1  "),
        SyncMethodsExample thread2 = new SyncMethodsExample("thread2: "),
        thread1 start();
        thread2 start(),
        boolean t1IsAlive = true;
        boolean t2IsAlive = true,
        do {
            if (t1IsAlive &amp,&amp, !thread1 isAlive()) {
                t1IsAlive = false,
                System out println("t1 is dead "),}
            if (t2IsAlive &amp,&amp, !thread2 isAlive()) {
                t2IsAlive = false,
                System out println("t2 is dead "),}
        } while (t1IsAlive || t2IsAlive),}
    void randomWait() {
        try {
            Thread currentThread() sleep((long) (3000 * Math.random()));
        } catch (InterruptedException e) {
            System out println("Interrupted!"),
        }       }
    public synchronized void run() {
        SynchronizedOutput displayList(getName(), msg),
    } }
class SynchronizedOutput {
    // if the 'synchronized' keyword is removed, the message is displayed in random fashion
    public static synchronized void displayList(String name, String list[]) {
        for (int i = 0; i &lt, list length, i++) {
        SyncMethodsExample t = (SyncMethodsExample) Thread currentThread(),
            t randomWait(),
            System out println(name + list[i]),
        }       }}
```

SYNCHRONIZED BLOCKS

Static methods synchronize on the class lock Acquiring and relinquishing a class lock by a thread in order to execute a static synchronized method, proceeds analogous to that of an object lock for a synchronized instance method A thread acquires the class lock before it can proceed with the execution of any static synchronized method in the class, blocking other threads wishing to execute any such methods in the same class This, of course, does not apply to static, non-synchronized methods, which can be invoked at any time Synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass The synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object The general form of the synchronized block is as follows:

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

A compile-time error occurs if the expression produces a value of any primitive type If execution of the block completes normally, then the lock is released If execution of the block completes abruptly, then the lock is released. A thread can hold more than one lock at a time Synchronized statements can be nested Synchronized statements with identical expressions can be nested The expression must evaluate to a non-null reference value, otherwise, a NullPointerException is thrown. The code block is usually related to the object on which the synchronization is being done This is the case with synchronized methods, where the execution of the method is synchronized on the lock of the current object:

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

Once a thread has entered the code block after acquiring the lock on the specified object, no other thread will be able to execute the code block, or any other code requiring the same object lock, until the lock is relinquished This happens when the execution of the code block completes normally or an uncaught exception is thrown Object specification in the synchronized statement is mandatory A class can choose to synchronize the execution of a part of a method, by using the this reference and putting the relevant part of the method in the synchronized block The braces of the block cannot be left out, even if the code block has just one statement

```
class SmartClient {
  BankAccount account,
 //
 public void updateTransaction() {
   synchronized (account) { // (1) synchronized block
    account update(), // (2)
   }
 }
```

In the previous example, the code at (2) in the synchronized block at (1) is synchronized on the BankAccount object If several threads were to concurrently execute the method updateTransaction() on an object of SmartClient, the statement at (2) would be executed by one thread at a time, only after synchronizing on the BankAccount object associated with this particular instance of SmartClient. Inner classes can access data in their enclosing context. An inner object might need to synchronize on its associated outer object, in order to ensure integrity of data in the latter. This is illustrated in the following code where the synchronized block at (5) uses the special form of the this reference to synchronize on the outer object associated with an object of the inner class This setup ensures that a thread executing the method setPi() in an inner object can only access the private double field myPi at (2) in the synchronized block at (5), by first acquiring the lock on the associated outer object. If another thread has the lock of the associated outer object, the thread in the inner object has to wait for the lock to be relinquished before it can proceed with the execution of the synchronized block at (5). However, synchronizing on an inner object and on its associated outer object are independent of each other, unless enforced explicitly, as in the following code

```
class Outer { // (1) Top-level Class
  private double myPi, // (2)
  protected class Inner { // (3) Non-static member Class
    public void setPi() { // (4)
      synchronized(Outer this) { // (5) Synchronized block on outer object
        myPi = Math PI, // (6)
      }
    }
  }
}
```

Below example shows how synchronized block and object locks are used to coordinate access to shared objects by multiple threads

```
public class SyncBlockExample extends Thread {
        static String[] msg = { "Beginner", "java", "tutorial,", " ,", "com", "is", "the", "best" },
        public SyncBlockExample(String id) {
                super(id),
        }
        public static void main(String[] args) {
                SyncBlockExample thread1 = new SyncBlockExample("thread1  ");
                SyncBlockExample thread2 = new SyncBlockExample("thread2: ");
                thread1 start(),
                thread2 start();
                boolean t1IsAlive = true;
                boolean t2IsAlive = true,
                do {
```

```
            if (t1IsAlive &amp;amp;&amp;amp; !thread1.isAlive()) {
                t1IsAlive = false;
                System.out.println("t1 is dead ");
            }
            if (t2IsAlive &amp;amp;&amp;amp; !thread2.isAlive()) {
                t2IsAlive = false;
                System.out.println("t2 is dead ");
            }
        } while (t1IsAlive || t2IsAlive);}
    void randomWait() {
        try {
            Thread.currentThread().sleep((long) (3000 * Math.random()));
        } catch (InterruptedException e) {
            System.out.println("Interrupted!");
        }    }
    public void run() {
        synchronized (System.out) {
            for (int i = 0; i &lt; msg.length; i++) {
                randomWait();
                System.out.println(getName() + msg[i]);
            }
        }    }}
```

Synchronized blocks can also be specified on a class lock

synchronized (<class name>.class) { <code block> }

The block synchronizes on the lock of the object denoted by the reference <class name>.class A static synchronized method classAction() in class A is equivalent to the following declaration

```
static void classAction() {
  synchronized (A.class) { // Synchronized block on class A
    //    } }
```

In summary, a thread can hold a lock on an object

- by executing a synchronized instance method of the object
- by executing the body of a synchronized block that synchronizes on the object
- by executing a synchronized static method of a class

THREAD STATES

A Java thread is always in one of several states which could be running, sleeping, dead, etc A thread can be in any of the following states

- New Thread state (Ready-to-run state)
- Runnable state (Running state)
- Not Runnable state
- Dead state

NEW THREAD

A thread is in this state when the instantiation of a Thread object creates a new thread but does not start it running A thread starts life in the Ready-to-run state You can call only the start() or stop() methods when the thread is in this state Calling any method besides start() or stop() causes an IllegalThreadStateException

RUNNABLE

When the start() method is invoked on a New Thread() it gets to the runnable state or running state by calling the run() method A Runnable thread may actually be running, or may be awaiting its turn to run

NOT RUNNABLE

A thread becomes Not Runnable when one of the following four events occurs.

- When sleep() method is invoked and it sleeps for a specified amount of time
- When suspend() method is invoked
- When the wait() method is invoked and the thread waits for notification of a free resource or waits for the completion of another thread or waits to acquire a lock of an object
- The thread is blocking on I/O and waits for its completion

Example. Thread.currentThread() sleep(1000);

Note Thread currentThread() may return an output like Thread[threadA,5,main]

The output shown in bold describes

- the name of the thread,
- the priority of the thread, and
- the name of the group to which it belongs

Here, the run() method put itself to sleep for one second and becomes Not Runnable during that period A thread can be awakened abruptly by invoking the interrupt() method on the sleeping thread object or at the end of the period of time for sleep is over Whether or not it will actually start running depends on its priority and the availability of the CPU

Hence I hereby list the scenarios below to describe how a thread switches form a non runnable to a runnable state·

- If a thread has been put to sleep, then the specified number of milliseconds must elapse (or it must be interrupted).
- If a thread has been suspended, then its resume() method must be invoked
- If a thread is waiting on a condition variable, whatever object owns the variable must relinquish it by calling either notify() or notifyAll()
- If a thread is blocked on I/O, then the I/O must complete.

DEAD STATE

A thread enters this state when the run() method has finished executing or when the stop() method is invoked. Once in this state, the thread cannot ever run again

THREAD PRIORITY

In Java we can specify the priority of each thread relative to other threads Those threads having higher priority get greater access to available resources then lower priority threads. A Java thread inherits its priority from the thread that created it Heavy reliance on thread priorities for the behavior of a program can make the program non portable across platforms, as thread scheduling is host platform–dependent You can modify a thread's priority at any time after its creation using the setPriority() method and retrieve the thread priority value using getPriority() method

The following static final integer constants are defined in the Thread class:
MIN_PRIORITY (0) Lowest Priority NORM_PRIORITY (5) Default Priority MAX_PRIORITY (10) Highest Priority

- Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened
- The waiting thread times out
- Another thread interrupts the waiting thread.

Notify

- Invoking the notify() method on an object wakes up a single thread that is waiting on the lock of this object.
- A call to the notify() method has no consequences if there are no threads in the wait set of the object
- The notifyAll() method wakes up all threads in the wait set of the shared object.

Below program shows three threads, manipulating the same stack Two of them are pushing elements on the stack, while the third one is popping elements off the stack This example illustrates how a thread waiting as a result of calling the wait() method on an object, is notified by another thread calling the notify() method on the same object

THREAD SCHEDULER

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

Preemptive scheduling: If a thread with a higher priority than all other Runnable threads becomes Runnable, the scheduler will preempt the running thread (is moved to the runnable state) and choose the new higher priority thread for execution.

Time-Slicing or Round-Robin scheduling: A running thread is allowed to execute for a fixed length of time (a time slot it's assigned to), after which it moves to the Ready-to-run state (runnable) to await its turn to run again A thread scheduler is implementation and platform-dependent, therefore, how threads will be scheduled is unpredictable across different platforms

YIELDING

A call to the static method yield(), defined in the Thread class, will cause the current thread in the Running state to move to the Runnable state, thus relinquishing the CPU. The thread is then at the mercy of the thread scheduler as to when it will run again. If there are no threads waiting in the Ready-to-run state, this thread continues execution. If there are other threads in the Ready-to-run state, their priorities determine which thread gets to execute. The yield() method gives other threads of the same priority a chance to run. If there are no equal priority threads in the "Runnable" state, then the yield is ignored

SLEEPING AND WAKING UP

The thread class contains a static method named sleep() that causes the currently running thread to pause its execution and transit to the Sleeping state. The method does not relinquish any lock that the thread might have. The thread will sleep for at least the time specified in its argument, before entering the runnable state where it takes its turn to run again If a thread is interrupted while sleeping, it will throw an InterruptedException when it awakes and gets to execute The Thread class has several overloaded versions of the sleep() method

WAITING AND NOTIFYING

Waiting and notifying provide means of thread inter-communication that synchronizes on the same object The threads execute wait() and notify() (or notifyAll()) methods on the shared object for this purpose The notifyAll(), notify() and wait() are methods of the Object class These methods can be invoked only from within a synchronized context (synchronized method or synchronized block), otherwise, the call will result in an IllegalMonitorStateException. The notifyAll() method wakes up all the threads waiting on the resource In this situation, the awakened threads compete for the resource One thread gets the resource and the others go back to waiting.

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

The wait() call can specify the time the thread should wait before being timed out An another thread can invoke an interrupt() method on a waiting thread resulting in an InterruptedException This is a checked exception and hence the code with the wait() method must be enclosed within a try catch block

notify() method signatures
final void notify()
final void notifyAll()

A thread usually calls the wait() method on the object whose lock it holds because a condition for its continued execution was not met. The thread leaves the Running state and transits to the Waiting-for-notification state There it waits for this condition to occur. The thread relinquishes ownership of the object lock The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object after acquiring its lock. The wait() method causes the current thread to wait until another thread notifies it of a condition change A thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents.

- Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened
- The waiting thread times out
- Another thread interrupts the waiting thread

Notify

- Invoking the notify() method on an object wakes up a single thread that is waiting on the lock of this object
- A call to the notify() method has no consequences if there are no threads in the wait set of the object
- The notifyAll() method wakes up all threads in the wait set of the shared object

Below program shows three threads, manipulating the same stack. Two of them are pushing elements on the stack, while the third one is popping elements off the stack. This example illustrates how a thread waiting as a result of calling the wait() method on an object, is notified by another thread calling the notify() method on the same object

```
class StackClass {

        private Object[] stackArray,
        private volatile int topOfStack,
        StackClass(int capacity) {
                stackArray = new Object[capacity],
                topOfStack = -1,
        }
        public synchronized Object pop() {
                System out println(Thread currentThread() + "  popping"),
                while (isEmpty()) {
                        try {
                                System out println(Thread.currentThread() + "  waiting to pop"),
                                wait(),
                        } catch (InterruptedException e) {
                                e printStackTrace(),
                        }
                }
                Object obj = stackArray[topOfStack],
                stackArray[topOfStack--] = null,
                System out println(Thread currentThread()+ ". notifying after pop");
                notify(),
```

```
            return obj,
        }
        public synchronized void push(Object element) {
            System.out println(Thread currentThread() + ". pushing");
            while (isFull()) {
                try {
                    System.out println(Thread currentThread() + "  waiting to push");
                    wait();
                } catch (InterruptedException e) {
                    e printStackTrace();
                }
            }
            stackArray[++topOfStack] = element;
            System out println(Thread currentThread() + "  notifying after push"),
            notify(),
        }
        public boolean isFull() {
            return topOfStack &gt;= stackArray.length - 1;
        }
        public boolean isEmpty() {
            return topOfStack &lt, 0,
        }
}

abstract class StackUser extends Thread {

        protected StackClass stack;
        StackUser(String threadName, StackClass stack) {
            super(threadName),
            this stack = stack,
            System out.println(this),
            setDaemon(true),
            start(),
        }
}
class StackPopper extends StackUser { // Stack Popper
        StackPopper(String threadName, StackClass stack) {
            super(threadName, stack),
        }
        public void run() {
            while (true) {
                stack pop(),
            }
        }
}
```

```
class StackPusher extends StackUser { // Stack Pusher
        StackPusher(String threadName, StackClass stack) {
                super(threadName, stack),
        }
        public void run() {
                while (true) {
                        stack push(new Integer(1)),
                }
        }
}
public class WaitAndNotifyExample {
        public static void main(String[] args) {
                StackClass stack = new StackClass(5),
                new StackPusher("One", stack),
                new StackPusher("Two", stack),
                new StackPopper("Three", stack),
                System out println("Main Thread sleeping "),
                try {
                        Thread sleep(500);
                } catch (InterruptedException e) {
                        e printStackTrace(),
                }
                System out println("Exit from Main Thread "),
        }
}
```

The field topOfStack in class StackClass is declared volatile, so that read and write operations on this variable will access the master value of this variable, and not any copies, during runtime Since the threads manipulate the same stack object and the push() and pop() methods in the class StackClassare synchronized, it means that the threads synchronize on the same object

How the program uses wait() and notify() for inter thread communication

(1) The synchronized pop() method – When a thread executing this method on the StackClass object finds that the stack is empty, it invokes the wait() method in order to wait for some other thread to fill the stack by using the synchronized push Once an other thread makes a push, it invokes the notify method

(2) The synchronized push() method – When a thread executing this method on the StackClass object finds that the stack is full, i t invokes the wait() method to await some other thread to remove an element to provide space for the newly to be pushed element Once another thread makes a pop, it invokes the notify method

JOINING

A thread invokes the join() method on another thread in order to wait for the other thread to complete its execution. Consider a thread t1 invokes the method join() on a thread t2. The join() call has no effect if thread t2 has already completed. If thread t2 is still alive, then thread t1 transits to the Blocked-for-join-completion state. Below is a program showing how threads invoke the overloaded thread join method

```
public class ThreadJoinDemo {

        public static void main(String[] args) {
                Thread t1 = new Thread("T1"),
                Thread t2 = new Thread("T2");
                try {
                        System out.println("Wait for the child threads to finish "),
                        t1.join(),
                        if (!t1.isAlive())
                                System out println("Thread T1 is not alive."),
                        t2 join(),
                        if (!t2.isAlive())
                                System.out println("Thread T2 is not alive."),
                } catch (InterruptedException e) {
                        System out.println("Main Thread interrupted."),
                }
                System.out println("Exit from Main Thread.");
        }
}
```

Output

```
Wait for the child threads to finish
Thread T1 is not alive
Thread T2 is not alive
Exit from Main Thread.
```

DEADLOCK

There are situations when programs become deadlocked when each thread is waiting on a resource that cannot become available The simplest form of deadlock is when two threads are each waiting on a resource that is locked by the other thread Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever in the Blocked-for-lock-acquisition state. The threads are said to be deadlocked Thread t1 at tries to synchronize first on string o1 and then on string o2. The thread t2 does the opposite It synchronizes first on string o2 then on string o1. Hence a deadlock can occur as explained above Below is a program that illustrates deadlocks in multithreading applications

```
public class DeadLockExample {
        String o1 = "Lock ",
        String o2 = "Step ",
```

```
Thread t1 = (new Thread("Printer1") {
        public void run() {
                while (true) {
                        synchronized (o1) {
                                synchronized (o2) {
                                        System out.println(o1 + o2),
                                }}}}    }),
Thread t2 = (new Thread("Printer2") {
        public void run() {
                while (true) {
                        synchronized (o2) {
                                synchronized (o1) {
                                        System out.println(o2 + o1),
                                }}}}    });
public static void main(String[] args) {
        DeadLockExample dLock = new DeadLockExample(),
        dLock.t1.start(),
        dLock.t2 start(),
}}
```

Note The following methods namely join, sleep and wait name the InterruptedException in its throws clause and can have a timeout argument as a parameter The following methods namely wait, notify and notifyAll should only be called by a thread that holds the lock of the instance on which the method is invoked The Thread start method causes a new thread to get ready to run at the discretion of the thread scheduler The Runnable interface declares the run method The Thread class implements the Runnable interface Some implementations of the Thread yield method will not yield to a thread of lower priority A program will terminate only when all user threads stop running A thread inherits its daemon status from the thread that created it

Review Questions

1 Explain about life cycle of a thread
2 Describe how multithreading is implemented with examples.
3. Discuss the thread scheduling.
4 List some of the thread functions with example

CHAPTER 12

FILES STREAM

The java io package contains nearly every class you might ever need to perform input and output (I/O) in Java All these streams represent an input source and an output destination. The stream in the java io package supports many data such as primitives, object, localized characters, etc

Stream

A stream can be defined as a sequence of data There are two kinds of Streams –

- InPutStream – The InputStream is used to read data from a source
- OutPutStream – The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O We will see the most commonly used examples one by one –

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream Following is an example which makes use of these two classes to copy an input file into an output file –

```
import java io *,
public class CopyFile {
public static void main(String args[]) throws IOException {
    FileInputStream in = null,
    FileOutputStream out = null,
    try {
      in = new FileInputStream("input txt"),
      out = new FileOutputStream("output txt"),
      int c;
      while ((c = in read()) != -1) {
        out write(c),
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close(),        }
    }  }}
```

Now let's have a file input txt with the following content –

As a next step, compile the above program and execute it, which will result in creating output txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following –

```
$javac CopyFile java
$java CopyFile
```

Character Streams

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time. We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file –

```
import java io *;
public class CopyFile {
  public static void main(String args[]) throws IOException {
    FileReader in = null;
    FileWriter out = null,
    try {
      in = new FileReader("input txt"),
      out = new FileWriter("output txt"),
      int c,
      while ((c = in.read()) != -1) {
        out write(c),
      }
    }finally {
      if (in != null) {
        in.close(),
      }
      if (out != null) {
        out close(),
      }     }  }}
```

Now let's have a file input txt with the following content –

As a next step, compile the above program and execute it, which will result in creating output txt file with the same content as we have in input txt So let's put the above code in CopyFile java file and do the following –

```
$javac CopyFile java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen

- Standard Input – This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System in
- Standard Output – This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as System out
- Standard Error – This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System err

Following is a simple program, which creates InputStreamReader to read standard input stream until the user types a "q" –

```
import java io.*;
public class ReadConsole {
  public static void main(String args[]) throws IOException {
    InputStreamReader cin = null,
    try {
      cin = new InputStreamReader(System.in),
      System out.println("Enter characters, 'q' to quit.");
      char c,
      do {
        c = (char) cin read(),
        System.out print(c),
      } while(c != 'q');
    }finally {
      if (cin != null) {
        cin close();
      }     }  }}
```

Let's keep the above code in ReadConsole java file and try to compile and execute it as shown in the following program This program continues to read and output the same character until we press 'q' –

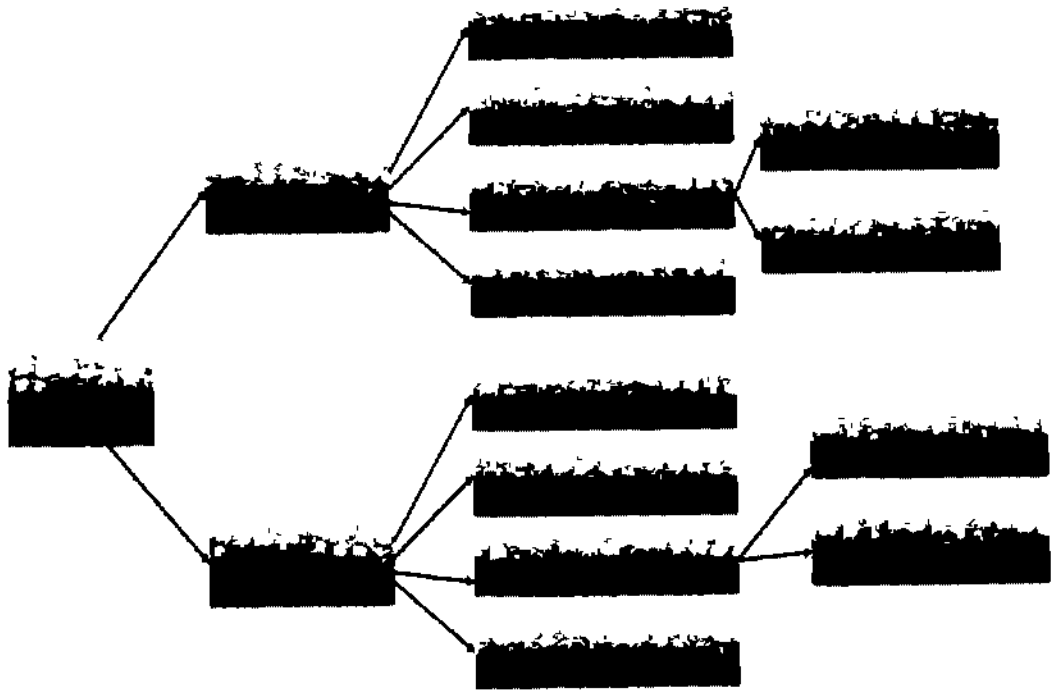
OUTPUT

```
Enter characters, 'q' to quit
1
1
e
e
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available. Following constructor takes a file name as a string to create an input stream object to read the file –

```
InputStream f = new FileInputStream("C:/java/hello"),
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows –

```
File f = new File("C /java/hello"),
InputStream f = new FileInputStream(f),
```

Once you have InputStream object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr.No	Method & Description
	public void close() throws IOException{}
1	This method closes the file output stream Releases any system resources associated with the file Throws an IOException
	protected void finalize()throws IOException {}
2	This method cleans up the connection to the file Ensures that the close method of this file output stream is called when there are no more references to this stream Throws an IOException
	public int read(int r)throws IOException{}
3	This method reads the specified byte of data from the InputStream Returns an int Returns the next byte of data and -1 will be returned if it's the end of the file
	public int read(byte[] r) throws IOException{}
4	This method reads r length bytes from the input stream into an array Returns the total number of bytes read If it is the end of the file, -1 will be returned
	public int available() throws IOException{}
5	Gives the number of bytes that can be read from this file input stream Returns an int

There are other important input streams available, for more detail you can refer to the following links −

- ByteArrayInputStream
- DataInputStream

FileOutputStream

FileOutputStream is used to create a file and write data into it The stream would create a file, if it doesn't already exist, before opening it for output Here are two constructors which can be used to create a FileOutputStream object Following constructor takes a file name as a string to create an input stream object to write the file −

```
OutputStream f = new FileOutputStream("C /java/hello")
```

Following constructor takes a file object to create an output stream object to write the file First, we create a file object using File() method as follows –

File f = new File("C /java/hello"),
OutputStream f = new FileOutputStream(f),

Once you have OutputStream object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream

Sr.No.	Method & Description
1	public void close() throws IOException{} This method closes the file output stream Releases any system resources associated with the file Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file Ensures that the close method of this file output stream is called when there are no more references to this stream Throws an IOException
3	public void write(int w)throws IOException{} This methods writes the specified byte to the output stream
4	public void write(byte[] w) Writes w length bytes from the mentioned byte array to the OutputStream

There are other important output streams available, for more detail you can refer to the following links –

- ByteArrayOutputStream
- DataOutputStream

Example
Following is the example to demonstrate InputStream and OutputStream –

```
import java.io *,
public class fileStreamTest {
  public static void main(String args[]) {
    try {
      byte bWrite [] = {11,21,3,40,5},
      OutputStream os = new FileOutputStream("test txt"),
      for(int x = 0; x < bWrite.length , x++) {
```

```
        os write( bWrite[x] );   // writes the bytes
      }
      os.close(),
      InputStream is = new FileInputStream("test txt");
      int size = is available(),
      for(int i = 0; i < size; i++) {
        System out print((char)is read() + "  ");
      }
      is.close();
    }catch(IOException e) {
      System out print("Exception"),
    }
  }
}
```

The above code would create file test txt and would write given numbers in binary format. Same would be the output on the stdout screen

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java

A directory is a File which can contain a list of other files and directories You use File object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories

There are two useful File utility methods, which can be used to create directories –

- The mkdir( ) method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet
- The mkdirs() method creates both a directory and all the parents of the directory

Following example creates "/tmp/user/java/bin" directory –

Example

```
import java.io File,
public class CreateDir {
  public static void main(String args[]) {
    String dirname = "/tmp/user/java/bin",
    File d = new File(dirname),
    // Create directory now
    d mkdirs();
  }
}
```

Compile and execute the above code to create "/tmp/user/java/bin"

Note – Java automatically takes care of path separators on UNIX and Windows as per conventions If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly

Listing Directories

You can use list( ) method provided by File object to list down all the files and directories available in a directory as follows –

Example

```
import java io File,
public class ReadDir {
  public static void main(String[] args) {
    File file = null,
    String[] paths,
    try {
      // create new file object
      file = new File("/tmp"),
      // array of files and directory
      paths = file list(),
      // for each name in the path array
      for(String path paths) {
        // prints filename and directory name
        System out println(path),
      }
    }catch(Exception e) {
      // if any error occurs
      e printStackTrace(),
    }
  }
}
```

This will produce the following result based on the directories and files available in your /tmp directory –

Output

```
test1 txt
test2 txt
ReadDir java
ReadDir class
```

Review Questions

1. Explain i/o stream and file stream in detail
2. Describe how files are implemented with examples
3. Discuss about directories
4. List some of the stream functions with example

CHAPTER 13

HTML AND ITS TAGS

What is HTML?

HTML stands for Hypertext Markup Language, and it is the most widely used language to write Web Pages

- Hypertext refers to the way in which Web pages (HTML documents) are linked together. Thus, the link available on a webpage is called Hypertext
- As its name suggests, HTML is a Markup Language which means you use HTML to simply "mark-up" a text document with tags that tell a Web browser how to structure it to display

Originally, HTML was developed with the intent of defining the structure of documents like headings, paragraphs, lists, and so forth to facilitate the sharing of scientific information between researchers. Now, HTML is being widely used to format web pages with the help of different tags available in HTML language

- HTML is the standard markup language for creating Web pages
- HTML stands for Hyper Text Markup Language
- HTML describes the structure of Web pages using markup
- HTML elements are the building blocks of HTML pages
- HTML elements are represented by tags
- HTML tags label pieces of content such as "heading", "paragraph", "table", and so on
- Browsers do not display the HTML tags, but use them to render the content of the page

Example	Example Explained
<!DOCTYPE html>	The <!DOCTYPE html> declaration defines this document to be HTML5
<html>	The <html> element is the root element of an HTML page
<head>	The <head> element contains meta information about the document
<title>Page Title</title>	The <title> element specifies a title for the document
</head>	
<body>	The <body> element contains the visible page content
My First Heading	The <h1> element defines a large heading
My first paragraph	The element defines a paragraph
</body>	
</html>	

HTML Tags

HTML tags are element names surrounded by angle brackets **<tagname>content goes here...</tagname>**

- HTML tags normally come in pairs like and
- The first tag in a pair is the start tag, the second tag is the end tag
- The end tag is written like the start tag, but with a **forward slash** inserted before the tag name
- The start tag is also called the opening tag, and the end tag the closing tag

<!DOCTYPE> Declaration

- The <!DOCTYPE> declaration represents the document type, and helps browsers to display web pages correctly
- It must only appear once, at the top of the page (before any HTML tags)
- The <!DOCTYPE> declaration is not case sensitive
- The <!DOCTYPE> declaration for HTML5 is <!DOCTYPE html>

HTML Basic Examples

HTML Documents

All HTML documents must start with a document type declaration: **<!DOCTYPE html>**
The HTML document itself begins with **<html>** and ends with **</html>**
The visible part of the HTML document is between **<body>** and **</body>**

Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Heading</h1>
<p>My first paragraph.</p>
</body>
</html>
```

HTML Headings

HTML headings are defined with the **<h1>** to **<h6>** tags.
<h1> defines the most important heading <h6> defines the least important heading

Example

```
<h1>This is heading 1</h1>
<h2>This is heading 2</h2>
<h3>This is heading 3</h3>
```

output

This is heading 1

This is heading 2

This is heading 3

HTML Paragraphs: ***HTML paragraphs are defined with the <p> tag:***

Example

```
<p>This is a paragraph </p>
<p>This is another paragraph.</p>
```

HTML Links: ***HTML links are defined with the <a> tag:***

Example

```
<a href="https://www google com">This is a link</a>
```

The link's destination is specified in the **href attribute**
Attributes are used to provide additional information about HTML elements

HTML Images: ***HTML images are defined with the*** *<img>* ***tag.***
The source file (src), alternative text (alt), width, and height are provided as attributes·
Example
<img src="pic jpg" alt="xxx com" width="104" height="142">

HTML Line Breaks: The HTML
 element defines a line break.
Use
 if you want a line break (a new line) without starting a new paragraph.
Example
This is
a paragraph
with line breaks </p›

The HTML Style Attribute: Setting the style of an HTML element, can be done with the style attribute The HTML style attribute has the following syntax:
<tagname style="*property value,*">
The **property** is a CSS property The **value** is a CSS value

HTML Background Color: The background-color property defines the background color for an HTML element This example sets the background color for a page to powderblue
Example
<body style="background-color powderblue,">
</body>

HTML Text Color The color property defines the text color for an HTML element
Example
<h1 style="color blue;">This is a heading</h1>
This is a paragraph

HTML Fonts: The font-family property defines the font to be used for an HTML element:
Example
<h1 style="font-family verdana,">This is a heading</h1>
This is a paragraph

HTML Text Size· The font-size property defines the text size for an HTML element.
Example
<h1 style="font-size 300%,">This is a heading</h1>
This is a paragraph.

HTML Text Alignment: The text-align property defines the horizontal text alignment for an HTML element· Example
<h1 style="text-align center;">Centered Heading</h1>
Centered paragraph

HTML Text Formatting: HTML Text Formatting Elements

Tag	Description
	Defines bold text
	Defines emphasized text
<i>	Defines italic text
<small>	Defines smaller text
	Defines important text
<sub>	Defines subscripted text
<sup>	Defines superscripted text
<ins>	Defines inserted text
<del>	Defines deleted text
<mark>	Defines marked/highlighted text

The HTML element defines bold text, without any extra importance
Example This text is bold
The HTML element defines strong text, with added semantic "strong" importance
Example This text is strong

HTML Attributes: ***Below is an alphabetical list of some attributes often used in HTML:***

Attribute	Description
Alt	Specifies an alternative text for an image, when the image cannot be displayed
disabled	Specifies that an input element should be disabled
Href	Specifies the URL (web address) for a link
Id	Specifies a unique id for an element
Src	Specifies the URL (web address) for an image
Style	Specifies an inline CSS style for an element
Title	Specifies extra information about an element (displayed as a tool tip)

HTML Table:
An HTML table is defined with the **<table>** tag.
Each table row is defined with the **<tr>** tag A table header is defined with the **<th>** tag By default, table headings are bold and centered A table data/cell is defined with the **<td>** tag
Example

```
<table style="width 100%">
  <tr> <th>Firstname</th>
    <th>Lastname</th>
    <th>Age</th></tr>
  <tr> <td>Jill</td>
    <td>Smith</td>
    <td>50</td> </tr>
  <tr>   <td>Eve</td>
    <td>Jackson</td>
    <td>94</td> </tr>
</table>
```

HTML Table - Adding a Border
If you do not specify a border for the table, it will be displayed without borders
A border is set using the CSS **border** property
Example

```
table, th, td {
   border  1px solid black,}
```

HTML Table - Adding Cell Padding
Cell padding specifies the space between the cell content and its borders
If you do not specify a padding, the table cells will be displayed without padding
To set the padding, use the CSS **padding** property
Example

```
th, td {   padding  15px,}
```

HTML Table - Left-align Headings
By default, table headings are bold and centered To left-align the table headings, use the CSS **text-align** property Example

```
th {
   text-align  left,
}
```

HTML Table - Adding Border Spacing
Border spacing specifies the space between the cells
To set the border spacing for a table, use the CSS **border-spacing** property
Example

```
table {
   border-spacing  5px,
}
```

HTML Table - Cells that Span Many Columns

To make a cell span more than one column, use the **colspan** attribute

Example

```
<table style="width.100%">
 <tr>
  <th>Name</th>
  <th colspan="2">Telephone</th>
 </tr>
 <tr>
  <td>Bill Gates</td>
  <td>55577854</td>
  <td>55577855</td>
 </tr>
</table>
```

HTML Table - Cells that Span Many Rows

To make a cell span more than one row, use the **rowspan** attribute

Example

```
<table style="width 100%">
 <tr>
  <th>Name </th>
  <td>Bill Gates</td>
 </tr>
 <tr>
  <th rowspan="2">Telephone </th>
  <td>55577854</td>
 </tr>
 <tr>
  <td>55577855</td>
 </tr>
</table>
```

HTML Table - Adding a Caption

To add a caption to a table, use the **<caption>** tag

Example

```
<table style="width 100%">
 <caption>Monthly savings</caption>
 <tr>  <th>Month</th>
  <th>Savings</th></tr>
 <tr>  <td>January</td>
  <td>$100</td>  </tr>
 <tr>  <td>February</td>
  <td>$50</td>  </tr>
</table>
```

HTML Table Tags:

Tag	Description
<table>	Defines a table
<th>	Defines a header cell in a table
<tr>	Defines a row in a table
<td>	Defines a cell in a table
<caption>	Defines a table caption
<colgroup>	Specifies a group of one or more columns in a table for formatting
<col>	Specifies column properties for each column within a <colgroup> element
<thead>	Groups the header content in a table
<tbody>	Groups the body content in a table
<tfoot>	Groups the footer content in a table

Ordered HTML List

An ordered list starts with the **<ol>** tag Each list item starts with the **<li>** tag.

Ordered HTML List - The Type Attribute: *The **type** attribute of the <ol> tag, defines the type of the list item marker:*

Type	Description
type="1"	The list items will be numbered with numbers (default)
type="A"	The list items will be numbered with uppercase letters
type="a"	The list items will be numbered with lowercase letters
type="I"	The list items will be numbered with uppercase roman numbers
type="i"	The list items will be numbered with lowercase roman numbers

```
<ol type="1">
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ol>
```

HTML Description Lists

HTML also supports description lists

A description list is a list of terms, with a description of each term.

The **<dl>** tag defines the description list, the **<dt>** tag defines the term (name), and the **<dd>** tag describes each term.

Example

```
<dl>
  <dt>Coffee</dt>
  <dd>- black hot drink</dd>
  <dt>Milk</dt>
  <dd>- white cold drink</dd>
</dl>
```

Horizontal Lists

HTML lists can be styled in many different ways with CSS. One popular way is to style a list horizontally, to create a menu

Example

```
<!DOCTYPE html>
<html>
<head>
<style>
ul {
   list-style-type: none,
   margin: 0,
   padding: 0,
   overflow: hidden,
   background-color: #333333,
}
li {
   float: left,
}
li a {
   display: block,
   color: white,
   text-align: center,
   padding: 16px,
   text-decoration: none,
}
li a hover {
   background-color: #111111,
}
```

```
</style>
</head>
<body>
<ul>
  <li><a href="#home">Home</a></li>
  <li><a href="#news">News</a></li>
  <li><a href="#contact">Contact</a></li>
  <li><a href="#about">About</a></li>
</ul>
</body>
</html>
```

FRAMES:

HTML frames are used to divide your browser window into multiple sections where each section can load a separate HTML document. A collection of frames in the browser window is known as a frameset The window is divided into frames in a similar way the tables are organized into rows and columns.

Disadvantages of Frames

There are few drawbacks with using frames, so it's never recommended to use frames in your webpages −

- Some smaller devices cannot cope with frames often because their screen is not big enough to be divided up
- Sometimes your page will be displayed differently on different computers due to different screen resolution
- The browser's *back* button might not work as the user hopes
- There are still few browsers that do not support frame technology

Creating Frames

To use frames on a page we use <frameset> tag instead of <body> tag The <frameset> tag defines, how to divide the window into frames The **rows** attribute of <frameset> tag defines horizontal frames and **cols** attribute defines vertical frames. Each frame is indicated by <frame> tag and it defines which HTML document shall open into the frame

Example

Following is the example to create three horizontal frames −

```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Frames</title>
  </head>
  <frameset rows = "10%,80%,10%">
    <frame name = "top" src = "/html/top_frame.htm" />
    <frame name = "main" src = "/html/main_frame htm" />
    <frame name = "bottom" src = "/html/bottom_frame htm" />
  <noframes>
      <body>Your browser does not support frames.</body>
```

```
    </noframes>
  </frameset>
</html>
```

Example

Let's put the above example as follows, here we replaced rows attribute by cols and changed their width This will create all the three frames vertically –

```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Frames</title>
  </head>
 <frameset cols = "25%,50%,25%">
    <frame name = "left" src = "/html/top_frame htm" />
    <frame name = "center" src = "/html/main_frame htm" />
    <frame name = "right" src = "/html/bottom_frame htm" />
    <noframes>
      <body>Your browser does not support frames </body>
    </noframes>
  </frameset>
</html>
```

Following are important attributes of the <frameset> tag –

Sr.No	Attribute & Description
1	**cols** Specifies how many columns are contained in the frameset and the size of each column You can specify the width of each column in one of the four ways – Absolute values in pixels For example, to create three vertical frames, use *cols = "100, 500. 100"* A percentage of the browser window For example, to create three vertical frames, use *cols = "10%, 80%, 10%"* Using a wildcard symbol For example, to create three vertical frames, use *cols = "10%, *, 10%"* In this case wildcard takes remainder of the window As relative widths of the browser window For example, to create three vertical frames, use *cols = "3*, 2*, 1*"* This is an alternative to percentages You can use relative widths of the browser window Here the window is divided into sixths the first column takes up half of the window, the second takes one third, and the third takes one sixth
2	**rows** This attribute works just like the cols attribute and takes the same values, but it is used to specify the rows in the frameset For example, to create two horizontal frames, use *rows = "10%, 90%"* You can specify the height of each row in the same way as explained above for columns
3	**border** This attribute specifies the width of the border of each frame in pixels For example,

border = "5" A value of zero means no border

4 **frameborder**
This attribute specifies whether a three-dimensional border should be displayed between frames This attribute takes value either 1 (yes) or 0 (no) For example frameborder = "0" specifies no border

5 **framespacing**
This attribute specifies the amount of space between frames in a frameset This can take any integer value For example framespacing = "10" means there should be 10 pixels spacing between each frames

6 **src**
This attribute is used to give the file name that should be loaded in the frame Its value can be any URL For example, src = "/html/top_frame.htm" will load an HTML file available in html directory

7 **name**
This attribute allows you to give a name to a frame It is used to indicate which frame a document should be loaded into. This is especially important when you want to create links in one frame that load pages into an another frame, in which case the second frame needs a name to identify itself as the target of the link

8 **frameborder**
This attribute specifies whether or not the borders of that frame are shown; it overrides the value given in the frameborder attribute on the <frameset> tag if one is given, and this can take values either 1 (yes) or 0 (no)

9 **marginwidth**
This attribute allows you to specify the width of the space between the left and right of the frame's borders and the frame's content The value is given in pixels For example marginwidth = "10"

10 **marginheight**
This attribute allows you to specify the height of the space between the top and bottom of the frame's borders and its contents. The value is given in pixels For example marginheight = "10"

11 **noresize**
By default, you can resize any frame by clicking and dragging on the borders of a frame The noresize attribute prevents a user from being able to resize the frame For example noresize = "noresize".

12 **scrolling**
This attribute controls the appearance of the scrollbars that appear on the frame This takes values either "yes", "no" or "auto" For example scrolling = "no" means it should not have scroll bars

13 **longdesc**
This attribute allows you to provide a link to another page containing a long description of the contents of the frame For example longdesc = "framedescription.htm"

Browser Support for Frames

If a user is using any old browser or any browser, which does not support frames then <noframes> element should be displayed to the user

So you must place a <body> element inside the <noframes> element because the <frameset> element is supposed to replace the <body> element, but if a browser does not understand <frameset> element then it should understand what is inside the <body> element which is contained in a <noframes> element

You can put some nice message for your user having old browsers For example, *Sorry!! your browser does not support frames* as shown in the above example

Frame's name and target attributes

One of the most popular uses of frames is to place navigation bars in one frame and then load main pages into a separate frame

Let's see following example where a test htm file has following code –

```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Target Frames</title>
  </head>
  <frameset cols = "200, *">
    <frame src = "/html/menu htm" name = "menu_page" />
    <frame src = "/html/main htm" name = "main_page" />
   <noframes>
      <body>Your browser does not support frames </body>
    </noframes>
  </frameset>
</html>
```

Here, we have created two columns to fill with two frames The first frame is 200 pixels wide and will contain the navigation menu bar implemented by **menu.htm** file The second column fills in remaining space and will contain the main part of the page and it is implemented by **main.htm** file For all the three links available in menu bar, we have mentioned target frame as **main_page**, so whenever you click any of the links in menu bar, available link will open in main page

Following is the content of menu htm file

```
<!DOCTYPE html>
<html>
  <body bgcolor = "#4a7d49">
    <a href = "http //www google com" target = "main_page">Google</a>
    <br />
    <br />
    <a href = "http·//www.microsoft com" target = "main_page">Microsoft</a>
    <br />
    <br />

    <a href = "http //news bbc co uk" target = "main_page">BBC News</a>
  </body>
```

```
</html>
```

Following is the content of main htm file −

```
<!DOCTYPE html>
<html>
  <body bgcolor = "#b5dcb3">
    <h3>This is main page and content from any link will be displayed here </h3>
    <p>So now click any link and see the result </p>
  </body>
</html>
```

The *targetattribute* can also take one of the following values −

Sr.No	Option & Description
1	**_self** Loads the page into the current frame
2	**_blank** Loads a page into a new browser window Opening a new window
3	**_parent** Loads the page into the parent window, which in the case of a single frameset is the main browser window
4	**_top** Loads the page into the browser window, replacing any current frames
5	**targetframe** Loads the page into a named targetframe

FORMS:

HTML Forms are required, when you want to collect some data from the site visitor For example, during user registration you would like to collect information such as name, email address, credit card, etc A form will take input from the site visitor and then will post it to a back-end application such as CGI, ASP Script or PHP script etc The back-end application will perform required processing on the passed data based on defined business logic inside the application There are various form elements available like text fields, textarea fields, drop-down menus, radio buttons, checkboxes, etc

The HTML **<form>** tag is used to create an HTML form and it has following syntax −

```
<form action = "Script URL" method = "GET|POST">
  form elements like input, textarea etc
</form>
```

Form Attributes

Apart from common attributes, following is a list of the most frequently used form attributes –

Sr.No	Attribute & Description
1	**Action** Backend script ready to process your passed data
2	**Method** Method to be used to upload data The most frequently used are GET and POST methods
3	**target** Specify the target window or frame where the result of the script will be displayed. It takes values like _blank, _self, _parent etc
4	**enctype** You can use the enctype attribute to specify how the browser encodes the data before it sends it to the server Possible values are – **application/x-www-form-urlencoded** – This is the standard method most forms use in simple scenarios. **mutlipart/form-data** – This is used when you want to upload binary data in the form of files like image, word file etc

HTML Form Controls

There are different types of form controls that you can use to collect data using HTML form –

- Text Input Controls
- Checkboxes Controls
- Radio Box Controls
- Select Box Controls
- File Select boxes
- Hidden Controls
- Clickable Buttons
- Submit and Reset Button

Text Input Controls

There are three types of text input used on forms –

- **Single-line text input controls** – This control is used for items that require only one line of user input, such as search boxes or names They are created using HTML **<input>** tag
- **Password input controls** – This is also a single-line text input but it masks the character as soon as a user enters it They are also created using HTMl <input> tag
- **Multi-line text input controls** – This is used when the user is required to give details that may be longer than a single sentence Multi-line input controls are created using HTML **<textarea>** tag

Single-line text input controls

This control is used for items that require only one line of user input, such as search boxes or names They are created using HTML <input> tag

Example

Here is a basic example of a single-line text input used to take first name and last name −

```
<!DOCTYPE html>
<html>
  <head>
    <title>Text Input Control</title>
  </head>
  <body>
    <form >
      First name. <input type = "text" name = "first_name" />
      <br>
      Last name  <input type = "text" name = "last_name" />
    </form>
  </body>
</html>
```

Following is the list of attributes for <input> tag for creating text field

Sr.No	Attribute & Description
1	**type** Indicates the type of input control and for text input control it will be set to **text.**
2	**name** Used to give a name to the control which is sent to the server to be recognized **and get** the value
3	**value** This can be used to provide an initial value inside the control.
4	**size** Allows to specify the width of the text-input control in terms of characters.
5	**maxlength** Allows to specify the maximum number of characters a user can enter into the text box.

Password input controls

This is also a single-line text input but it masks the character as soon as a user enters it They are also created using HTML <input>tag but type attribute is set to **password.**

Example

Here is a basic example of a single-line password input used to take user password −

```
<!DOCTYPE html>
<html>
  <head>
    <title>Password Input Control</title>
  </head>
  <body>
    <form >
      User ID . <input type = "text" name = "user_id" />
      <br>
```

```
    Password  <input type = "password" name = "password" />
  </form>
 </body>
</html>
```

Following is the list of attributes for <input> tag for creating password field

Sr.No	Attribute & Description
1	**type** Indicates the type of input control and for password input control it will be set to **password**
2	**name** Used to give a name to the control which is sent to the server to be recognized and get the value
3	**value** This can be used to provide an initial value inside the control
4	**size** Allows to specify the width of the text-input control in terms of characters
5	**maxlength** Allows to specify the maximum number of characters a user can enter into the text box

Multiple-Line Text Input Controls

This is used when the user is required to give details that may be longer than a single sentence Multi-line input controls are created using HTML <textarea> tag.

Example

Here is a basic example of a multi-line text input used to take item description −

```
<!DOCTYPE html>
<html>
  <head>
    <title>Multiple-Line Input Control</title>
  </head>
  <body>
    <form>
      Description  <br />
      <textarea rows = "5" cols = "50" name = "description">
        Enter description here
      </textarea>
    </form>
  </body>

</html>
```

This will produce the following result −

Attributes

Following is the list of attributes for <textarea> tag

Sr.No	Attribute & Description
1	**name** Used to give a name to the control which is sent to the server to be recognized and get the value
2	**rows** Indicates the number of rows of text area box
3	**cols** Indicates the number of columns of text area box

Checkbox Control

Checkboxes are used when more than one option is required to be selected They are also created using HTML <input> tag but type attribute is set to **checkbox.**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Checkbox Control</title>
  </head>
  <body>
    <form>
      <input type = "checkbox" name = "maths" value = "on"> Maths
      <input type = "checkbox" name = "physics" value = "on"> Physics
    </form>
  </body>
</html>
```

Following is the list of attributes for <checkbox> tag

Sr.No	Attribute & Description
1	**type** Indicates the type of input control and for checkbox input control it will be set to **checkbox.**
2	**name** Used to give a name to the control which is sent to the server to be recognized and get the value
3	**value** The value that will be used if the checkbox is selected
4	**checked** Set to *checked* if you want to select it by default

Radio Button Control

Radio buttons are used when out of many options, just one option is required to be selected They are also created using HTML <input> tag but type attribute is set to **radio**

Example

```
<html>
  <head>
    <title>Radio Box Control</title>
  </head>
  <body>
    <form>
      <input type = "radio" name = "subject" value = "maths"> Maths
      <input type = "radio" name = "subject" value = "physics"> Physics
    </form>
  </body>
</html>
```

Following is the list of attributes for radio button

Sr.No	Attribute & Description
1	**type** Indicates the type of input control and for checkbox input control it will be set to radio
2	**name** Used to give a name to the control which is sent to the server to be recognized and get the value
3	**value** The value that will be used if the radio box is selected
4	**checked** Set to *checked* if you want to select it by default

Select Box Control

A select box, also called drop down box which provides option to list down various options in the form of drop down list, from where a user can select one or more options

```
<html>
  <head>
    <title>Select Box Control</title>
  </head>
  <body>
    <form>
      <select name = "dropdown">
        <option value = "Maths" selected>Maths</option>
        <option value = "Physics">Physics</option>
      </select>
    </form>
  </body>
</html>
```

Following is the list of important attributes of <select> tag –

Sr.No	Attribute & Description
1	**name** Used to give a name to the control which is sent to the server to be recognized and get the value
2	**size** This can be used to present a scrolling list box
3	**multiple** If set to "multiple" then allows a user to select multiple items from the menu

Following is the list of important attributes of <option> tag –

Sr.No	Attribute & Description
1	**value** The value that will be used if an option in the select box box is selected
2	**selected** Specifies that this option should be the initially selected value when the page loads
3	**label** An alternative way of labeling options

Button Controls: There are various ways in HTML to create clickable buttons You can also create a clickable button using <input>tag by setting its type attribute to **button** The type attribute can take the following values –

Sr.No	Type & Description
1	**submit** This creates a button that automatically submits a form
2	**reset** This creates a button that automatically resets form controls to their initial values
3	**button** This creates a button that is used to trigger a client-side script when the user clicks that button
4	**image** This creates a clickable button but we can use an image as background of the button

```
<body>
  <form>
    <input type = "submit" name = "submit" value = "Submit" />
    <input type = "reset" name = "reset"  value = "Reset" />
  </form>  </body>
```

CHAPTER 14

JAVASCRIPT

JavaScript - History

JavaScript was designed to 'plug a gap' in the techniques available for creating web-pages HTML is relatively easy to learn, but it is static It allows the use of links to load new pages, images, sounds, etc , but it provides very little support for any other type of interactivity
To create dynamic material it was necessary to use either

- CGI (Common Gateway Interface) programs
  - Can be used to provide a wide range of interactive features, but
  - Run on the server, i e
    - A user-action causes a request to be sent over the internet from the client machine to the server
    - The server runs a CGI program that generates a new page, based on the information supplied by the client
    - The new page is sent back to the client machine and is loaded in place of the previous page

    Thus every change requires communication back and forth across the internet
  - Written in languages such as Perl, which are relatively difficult to learn

- Java applets
  - Run on the client, so there is no need to send information back and forth over the internet for every change, but
  - Written in Java, which is relatively difficult to learn

Netscape Corporation set out to develop a language that

- Provides dynamic facilities similar to those available using CGI programs and Java applets
- Runs on the Client
- Is relatively easy to learn and use

They came up with ***LiveScript***
Netscape subsequently teamed-up with Sun Microsystems (the company that developed Java) and produced ***JavaScript***
Javascript only runs on Netscape browsers (e g , Netscape Navigator) However, Microsoft soon developed a version of JavaScript for their Internet Explorer browser It is called ***JScript*** The two languages are almost identical, although there are some minor differences

Internet browsers such as Internet Explorer and Netscape Navigator provide a range of features that can be controlled using a suitable program For example, windows can be opened and closed, items can be moved around the page, colours can be changed, information can be read or modified, etc However, in order to do this you need to know what items the browser contains, what operations can be carried out on each item, and the format of the necessary commands
Therefore, in order to program internet browsers, you need to know

- How to program in a suitable language (e g , Javascript/JScript)
- The internal structure of the browser

Variables & Literals

A variable is a *container* which has a name We use variables to hold information that may change from one moment to the next while a program is running For example, a shopping website might use a variable called *total* to hold the total cost of the goods the customer has selected The amount stored in this variable may change as the customer adds more goods or discards earlier choices, but the name *total* stays the same Therefore we can find out the current total cost at any time by asking the program to tell us what is currently stored in *total*

A literal, by contrast, doesn't have a name - it only has a value. For example, we might use a literal to store the VAT rate, since this doesn't change very often. The literal would have a value of (e g) 0 21. We could then obtain the final cost to the customer in the following way·

VAT is equal to *total* x 0 2 final total is equal to *total* + *VAT*

JavaScript accepts the following types of variables:

Numeric Any numeric value, whether a whole number (an *integer*) or a number that includes a fractional part (a *real*), e.g , 12, 3 14159, etc

String A group of text characters, e.g , Ian, Macintosh G4, etc

Boolean A value which can only be either True or False

Note that.

- When a new variable is created (or *declared*) its name must be preceded by the word var
- The type of the variable is determined by the way it is declared
    - if it is enclosed within quotes, it's a string
    - if it is set to true or false (without quotes) it's a boolean
    - if it is a number (without quotes) it's numeric
- We refer to the equals sign as the *assignment operator* because we use it to assign values to variables,
- Variable names must begin with a letter or an underscore
- Variable names must not include spaces
- JavaScript is case-sensitive
- Reserved words (i.e , words which indicate an action or operation in JavaScript) cannot be used as variable names.

Operators

Operators are a type of command They perform operations on variables and/or literals and produce a result JavaScript understands the following operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

In addition, JavaScript understands the following operators

++	Increment	Increase value by 1
--	Decrement	Decrease value by 1
-	Negation	Convert positive to negative, or vice versa

Below is a full list of these 'combined' operators

+=	'becomes equal to itself plus'
-=	'becomes equal to itself minus'
*=	'becomes equal to itself multiplied by'
/=	'becomes equal to itself divided by'
%=	'becomes equal to the amount which is left when it is divided by'

You may find the descriptions helpful when trying to remember what each operator does For example,5 * = 3

Functions in JavaScript

In JavaScript, as in other languages, we can create *functions* A function is a kind of mini-program that forms part of a larger program

Functions

- consist of one or more *statements* (i e , lines of program code that perform some operation)
- are separated in some way from the rest of the program, for example, by being enclosed in curly brackets, { }
- are given a unique name, so that they can be *called* from elsewhere in the program

Functions are used

- Where the same operation has to be performed many times within a program Rather than writing the same code again and again, it can be written once as a function and used repeatedly For example request_confirmation_from_user

- To make it easier for someone else to understand your program Rather than writing long, rambling programs in which every single operation is listed in turn, it is usually better to divide programs up into small groups of related operations For example set_variables_to_initial_values welcome_user

In JavaScript, functions are created in the following way

```
function name()
{
        statement,
```

```
        statement,
        Statement
}
```

Note that all the statements except the last statement must be followed by semi-colons The last statement doesn't need one, but if you do put a semi-colon after the last statement it won't cause any problems Here is an example of a simple function

```
function initialiseVariables()
{
        itemsSold = 0,
        nettPrice = 0,
        priceAfterTax = 0
}
```

When called, this function will set the three variables itemsSold, nettPrice and priceAfterTax to zero. To run this function from somewhere else in a program, we would simply call it by name, e.g. initialiseVariables(),
Note that the name must be followed by a pair of brackets ' The purpose of these will become clear later

Functions can be called from within other functions
For example.

```
function sayGoodbye()
{
        alert("Goodbye!")
}

function sayHello()
{
        alert("Hi, there!"),
        sayGoodbye()
}
```

When the function sayHello() is called, it first displays an *alert* on the screen An alert is simply box containing some text and an 'OK' button that the user can press to make the box disappear when the text has been read In this case the box will contain the words "Hi, there!"
The sayHello() function then calls the function sayGoodbye(), which posts another alert saying "Goodbye".
Click here to see this example working.
Note that the function sayGoodbye() is written first Browsers interpet JavaScript code line-by-line, starting at the top, and some browsers will report an error if they find a reference to a

function before they find the function itself Therefore functions should be declared before the point in the program where they are used

Passing Parameters to Functions

Some functions perform a simple task for which no extra information is needed However, it is often necessary to supply information to a function so that it can carry out its task. For example, if we want to create a function which adds VAT to a price, we would have to tell the function what the price is

To do this we would pass the price into the function as a *parameter* Parameters are listed in between the brackets that follow the function name For example

```
function name(parameter_1, parameter_2)
{
        statement(s),
}
```

In this case two parameters are used, but it's possible to use more than this if necessary. The additional parameter names would simply be added on to the list of parameters inside the brackets, separated from one another by commas It's also possible to use just one prameter if that's all that is needed

Here's an example of a simple function that accepts a single parameter

```
function addVAT(price)
{
        price *= 1.21,
        alert(price)
}
```

This function accepts a parameter called price, multiplies it by 1 21 (i e., adds an extra 21% to it), and then displays the new value in an alert box

Returning values from Functions

Sometimes we also need to get some information back from a function For example, we might want to add VAT to a price and then, instead of just displaying the result, pass it back to the user or display it in a table To get information back from a function we do the following

```
function addVAT(price)
{
        price *= 1 21,
        return price
}
```

To call this function we would do the following

```
var newPrice = addVAT(nettPrice)
```

The value returned by the funtion will be stored in the variable newPrice. Therefore this funtion will have the effect of making newPrice equal to nettPrice multiplied by 1 21

JavaScript Comparison Operators

As well as needing to assign values to variables, we sometime need to compare variables or literals. We do this using *Comparison Operators* Comparison Operators compare two values and produce an output which is either true or false

If we compare them, there are two possible outcomes

- They have the same value
- They do not have the same value

Therefore, we can make statements like these·

- They are the same
- They are the different
- The first is larger than the second
- The first is smaller than the second

.. and then perform a comparison to determine whether the statement is true or false

The basic comparison operator is:

==

(i.e , two equals signs, one after the other with no space in between).

It means 'is equal to' Compare this with the *assignment operator*, =, which means 'becomes equal to' The assignment operator *makes* two things equal to one another, the comparison operator tests to see if they are *already* equal to one another.

Here's an example showing how the comparison operator might be used

examPasses == totalStudents

JavaScript understands the following comparison operators.

==	'is equal to'
!=	'is NOT equal to'
<	'is less than'
>	'is greater than'
<=	'is less than or equal to'
>=	'is greater than or equal to'

Arrays

An array is a set of variables (e g , strings or numbers) that are grouped together and given a single name. For example, an array could be used to hold a set of strings representing the names of a group of people. It might be called *people* and hold (say) four different strings, each representing the name of a person. Sarah Patrick Jane Tim

Items are held in an array in a particular order, usually the order in which they were added to the array. However, one of the great advantages of arrays is that the ordering can be changed in various ways For example, the array above could easily be sorted so that the names are arranged in alphabetical order

Creating Arrays

To create an array, a new Array object must be declared. This can be done in two ways

```
var myArray = new Array("Sarah","Patrick","Jane","Tim");
```

Or

```
var myArray = ["Sarah","Patrick","Jane","Tim"],
```

In the first example, the new Array object is declared explicitly
In the second example, the array is declared in much the same way a string might be declared, except that square brackets ([]) are used at the beginning and end instead of quote-marks The square brackets indicate to the JavaScript interpreter that this sequence of characters is being declared as an array

Arrays are often used to hold data typed-in or otherwise collected from a user Therefore, it may be necessary to create the array first and add data to it later An empty array may be created in the following way

```
var myArray = new Array(),
```

The array thus created has no elements at all, but elements can be added as necessary later. If it is not known exactly what data will be stored in the array, but the number of items is known, it may be appropriate to create an array of a specific size This may be done in the following way

```
var myArray = new Array(4);
```

The array thus created has four elements, all of which are empty. These empty elements can be filled with data later on

Viewing and Modifying Array Elements

Suppose an array has been created using the following code

```
var demoArray = new Array("Sarah","Patrick","Jane","Tim"),
```

The number of elements in the array can be determined using the length property For example

```
alert(demoArray length),
```

The entire contents of the array can be viewed using the valueOf() method For example

```
alert(demoArray valueOf())
```

This piece of code will display an alert showing the entire contents of the array demoArray, with the various elements separated by commas

The value of a particular element in the array can be obtained by using its position in the array as an index For example

```
var indexNumber = prompt ("Please enter a number between 0 and 3","");
alert("Element " + indexNumber + " = " + demoArray[indexNumber]),
```

This piece of code will prompt the user to enter a number between 0 and 3 (the elements in an array are numbered from zero, so the four elements in this array will be numbered 0, 1, 2 and 3) It will then display the corresponding element from the array

It is also possible to change the value of an array element using its position in the array as an index For example

```
var newValue = prompt("Please enter your name","");
demoArray[0] = newValue,
alert(demoArray valueOf()),
```

This piece of code will prompt the user to enter their name, then place this string into element 0 of the array, over-writing the string previously held in that element It will then display the modified array using the valueOf() method

Adding and Removing Elements

The length property of an array can be altered as well as read

- ***increasing*** the length property adds extra (empty) elements onto the end of the array
- ***decreasing*** the length property removes some of the existing elements from the end of the array

Consider the following examples

(1)
```
var currentLength = demoArray length;
demoArray[currentLength] = "Fred",
alert(demoArray.valueOf());
```

This piece of code first determines the number of elements in the array using the length property It then uses this information to identify the next element position *after* the end of the existing array and places the string "Fred" into that position, thus creating a new element. Finally, it displays the modified array using the valueOf() method described earlier

Note that it isn't necessary to add 1 to the value of length in order to identify the next position in the array This is because length indicates the actual number of elements, even though the elements are numbered from zero. For example, if an array has two elements, the value of length will be 2; however, those two elements will be numbered 0 and 1 Therefore, if length is used as an index, it will indicate the *third* element in the array, not the second one.

(2)
```
var currentLength = demoArray length,
demoArray.length = currentLength - 1,
alert(demoArray valueOf()),
```

This piece of code first determines the number of elements in the array using the length property It then resets length to one less than its previous value, thus removing the last element in the array Finally, it displays the modified array using the valueOf() method

There are also several methods that add and remove elements directly, some of which are listed below However, it should be noted that these methods only work with Netscape Navigator, so it is generally preferable to use the methods described above since they work with most browsers

push() Adds one or more elements onto the end of an array For example

```
var lastElement = demoArray push("Fred", "Lisa"),
```

This piece of code would add two new elements, "Fred" and "Lisa" onto the end of the array. The variable lastElement would contain the value of the last element added, in this case "Lisa"

pop() Removes the last element from the end of an array For example

```
var lastElement = demoArray pop(),
```

This piece of code would remove the last element from the end of the array and return it in the variable lastElement

unshift() Adds one or more new elements to the *beginning* of an array, shifting the existing elements up to make room Operates in a similar fashion to push(), above.

shift() Removes the first element from the *beginning* of an array, shifting the existing elements down to fill the space Operates in a similar fashion to pop(), above

Splitting and Concatenating Arrays

Arrays can be split and concatenated using the following methods

slice(x,y) Copies the elements between positions x and y in the source array into a new array For example

```
newArray = demoArray slice(0,2),
alert(newArray valueOf()),
```

This piece of code will copy elements 0 and 1 from the array called demoArray into a new array called newArray. It will then display the contents ofnewArray using the valueOf() method described earlier.

concat(array) Concatenates the specified array and the array to which it is applied into a new array For example

```
combinedArray = demoArray concat(newArray),
alert(combinedArray valueOf()),
```

This piece of code will concatenate demoArray and the new array created in the last example (newArray) to form another array calledcombinedArray It will then display the contents of combinedArray using the valueOf() method

Rearranging Array Elements:

The order of the elements in an array can be modified using the following methods

reverse() Reverses the order of the elements within an array For example

```
demoArray reverse(),
alert(demoArray valueOf()),
```

This piece of code will reverse the order of the elements in the array, then display the re-ordered array using the valueOf() method described earlier

sort() Sorts the elements within the array Unless otherwise specified, the elements will be sorted alphabetically For example·

```
demoArray sort(),
alert(demoArray valueOf()),
```

This piece of code will sort the elements of the array into alphabetical order, then display the re-ordered array using the valueOf() method described earlier

Although the sort() method normally sorts arrays alphabetically, it can be modified to sort in other ways This is done by creating a special function and passing the name of that function to the sort()method as a parameter, e g demoArray sort(bylength),

Multi-Dimensional Arrays

The arrays described so far are *One-Dimensional Arrays* They are effectively just lists Sometimes, however, we need to store information which has more than one dimension, for example, the scores in a game

Sarah	18
Patrick	16
Jane	12
Tim	13

To store data of this type we use multi-dimensional arrays In JavaScript this is done by using arrays as elements of other arrays For example, the game scores could be stored in the following way

- Each person's data would be stored in a separate, two-element array (one element for the name and one element for the score)
- These four arrays (one for each person) would then be stored in a four-element array

We could create such an array in the following way

```
var person1 = new Array("Sarah", 18), var person2 = new Array("Patrick", 16),
var person3 = new Array("Jane", 12),
var person4 = new Array("Tim", 13),
var scores = new Array (person1,person2,person3,person4),
```

To identify the individual elements within a multi-dimensional array, we use two index values, one after the other. The first refers to an element in the outer array (scores in this example), and the second refers to an element in the inner array (person1, person2, person3 or person4 in this example) For example

```
function displayMDarray()
{
 for(x = 0, x <=3, x++)
 {
  alert(scores[x][0] + " has " + scores[x][1] + " points"),
 }
}
```

In this example, the array is accessed using pairs of values in square brackets, e g scores[x][0] The value in the first pair of square-brackets indicates one of the four 'person' arrays. The value in the second pair of square-brackets indicates one of the elements within that 'person' array, either the name (0) or the score (1)

The variable called x is incremented from 0 to 3 using a for loop Therefore, x will point to a different one of the four 'person' arrays each time through the loop The second value, 0 or 1, then selects either the name or score from within that array

Review Questions

1 Explain different HTML forms implementation with examples Describe how files are implemented with examples
2 Discuss HTML Tables
3 List operators in javascript with example
4 Write a program for user authendication using html and javascript
5 Describe array and functions used within array

CHAPTER 15

DOCUMENT OBJECT MODEL

Introduction:

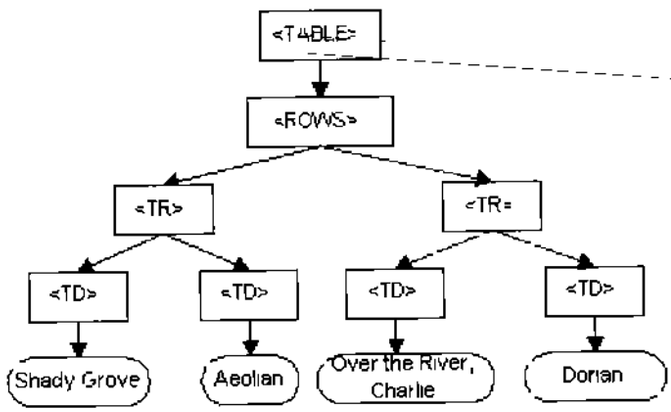
The Document Object Model (DOM) is a programming API for HTML and XML documents It defines the logical structure of documents and the way a document is accessed and manipulated In the DOM specification, the term "document" is used in the broad sense - increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data With the Document Object Model, programmers can create and build documents, navigate their structure, and add, modify, or delete elements and content Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions - in particular, the DOM interfaces for the internal subset and external subset have not yet been specified

What the Document Object Model is

The Document Object Model is a programming API for documents The object model itself closely resembles the structure of the documents it models For instance, consider this table, taken from an HTML document

```
<TABLE>
<ROWS>
<TR>
<TD>Shady Grove</TD>
<TD>Aeolian</TD>
</TR>
<TR>
<TD>Over the River, Charlie</TD>
<TD>Dorian</TD>
</TR>
</ROWS>
</TABLE>
```

The Document Object Model represents this table like this



DOM representation of the example table

In the Document Object Model, documents have a logical structure which is very much like a tree, to be more precise, it is like a "forest" or "grove" which can contain more than one tree However, the Document Object Model does not specify that documents be *implemented* as a tree or a grove , nor does it specify how the relationships among objects be implemented in any way In other words, the object model specifies the logical model for the programming interface, and this logical model may be implemented in any way that a particular implementation finds convenient In this specification, we use the term *structure model* to describe the tree-like representation of a document, we specifically avoid terms like "tree" or "grove" in order to avoid implying a particular implementation One important property of DOM structure models is *structural isomorphism* if any two Document Object Model implementations are used to create a representation of the same document, they will create the same structure model, with precisely the same objects and relationships The name "Document Object Model" was chosen because it is an "object model" is used in the traditional object oriented design sense documents are modeled using objects, and the model encompasses not only the structure of a document, but also the behavior of a document and the objects of which it is composed In other words, the nodes in the above diagram do not represent a data structure, they represent objects, which have functions and identity As an object model, the Document Object Model identifies

- the interfaces and objects used to represent and manipulate a document
- the semantics of these interfaces and objects - including both behavior and attributes
- the relationships and collaborations among these interfaces and objects

The structure of SGML documents has traditionally been represented by an abstract data model, not by an object model In an abstract data model, the model is centered around the data In object oriented programming languages, the data itself is encapsulated in objects which hide the data, protecting it from direct external manipulation The functions associated with these objects determine how the objects may be manipulated, and they are part of the object model

The Document Object Model currently consists of two parts, DOM Core and DOM HTML The DOM Core represents the functionality used for XML documents, and also serves as the basis for DOM HTML All DOM implementations must support the interfaces listed as "fundamental" in the Core specification, in addition, XML implementations must support the interfaces listed as "extended" in the Core specification The Level 1 DOM HTML specification defines additional functionality needed for HTML documents

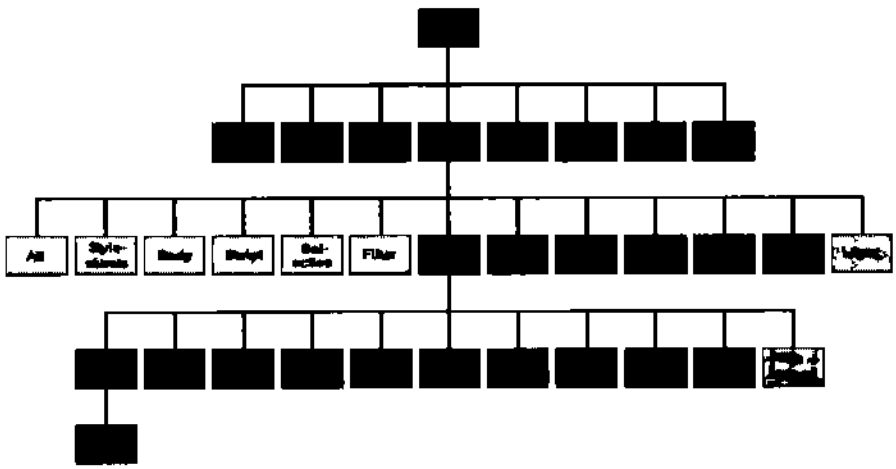
DOM Interfaces and DOM Implementations

The DOM specifies interfaces which may be used to manage XML or HTML documents It is important to realize that these interfaces are an abstraction - much like "abstract base classes" in C++, they are a means of specifying a way to access and manipulate an application's internal representation of a document In particular, interfaces do not imply a particular concrete implementation Each DOM application is free to maintain documents in any convenient representation, as long as the interfaces shown in this specification are supported Some DOM implementations will be existing programs that use the DOM interfaces to access software written long before the DOM specification existed Therefore, the DOM is designed to avoid implementation dependencies, in particular,

1 Attributes defined in the IDL do not imply concrete objects which must have specific data members - in the language bindings, they are translated to a pair of get()/set() functions, not to a data member (Read-only functions have only a get() function in the language bindings)
2 DOM applications may provide additional interfaces and objects not found in this specification and still be considered DOM compliant
3 Because we specify interfaces and not the actual objects that are to be created, the DOM can not know what constructors to call for an implementation In general, DOM users call the createXXX() methods on the Document class to create document structures, and DOM implementations create their own internal representations of these structures in their implementations of the createXXX() functions

Internet browsers contain many objects In the last few years the object structure of internet browsers has become standardised, making programming easier Prior to this, browsers from different manufacturers had different object structures Unfortunately, many such browsers are still in use

The objects are arranged into a hierarchy as shown below



The Document Object Model Objects shown in green are common to both Netscape Navigator and Internet Explorer, objects shown in yellow are found only in Internet Explorer while objects shown in blue are found only in Netscape Navigator The hierarchy of objects is known as the Document Object Model (DOM)

Its **properties** include.

status	The contents of the status bar (at the bottom of the browser window) For example window status = "Hi, there!", will display the string "Hi, there!" on the status bar Click here to see this line of code in operation
location	The location and URL of the document currently loaded into the window (as displayed in the location bar) For example alert(window location), will display an alert containing the location and URL of this document
length	The number of frames (if any) into which the current window is divided For example alert(window length), will display an alert indicating the number of frames in the current window
parent	The parent window, if the current window is a sub-window in a frameset For example var parentWindow = window parent, alert(parentWindow length), will place a string representing the parent window into the variable parentWindow, then use it to report the number of frames (if any) in the parent window
Top	The top-level window, of which all other windows are sub-windows For example var topWindow = window top, alert(topWindow length), will place a string representing the top-level window into the variable topWindow, then use it to report the number of frames (if any) in the top-level window top behaves in a very similar way to parent Where there are only two levels of windows, top and parent will both indicate the same window However, if there are more than two levels of windows, parent will indicate the parent of the current window, which may vary depending upon which window the code is in However, top will always indicate the very top-level window

Window **methods** include

alert()	Displays an 'alert' dialog box, containing text entered by the page designer, and an 'OK' button For example alert("Hi, there!"), will display a dialog box containing the message "Hi, there!"
confirm()	Displays a 'confirm' dialog box, containing text entered by the user, an 'OK' button, and a 'Cancel' button Returns true or false For example var response = confirm("Delete File?"), alert(response),

	will display a dialog box containing the message "Delete File?" along with an 'OK' button and a 'Cancel' button If the user clicks on 'OK' the variableresponse will contain the Boolean value true, and this will appear in the 'alert' dialog-box, If the user clicks on 'Cancel' the variable response will contain the Boolean value false and this will appear in the 'alert' dialog-box
prompt()	Displays a message, a box into which the user can type text, an 'OK' button, and a 'Cancel' button Returns a text string The syntax is prompt(message_string, default_response_string) For example var fileName = prompt("Select File", "file txt"), alert(fileName), will display a dialog box containing the message "Select File" along with an 'OK' button, a 'Cancel' button, and an area into which the user can type This area will contain the string "file txt", but this can be overwritten with a new name If the user clicks on 'OK' the variable fileName will contain the string "file txt" or whatever the user entered in its place, and this will be reported using an alert dialog box
open()	Opens a new browser window and loads either an existing page or a new document into it The syntax is open(URL_string, name_string, parameter_string) For example var parameters = "height=100,width=200", newWindow = open("05_JS4nw html", "newDocument", parameters), will open a new window 100 pixels high by 200 pixels wide An HTML document called '05_JS4nw html' will be loaded into this window
close()	Closes a window If no window is specified, closes the current window The syntax is window_name close() For example newWindow close() will close the new window opened by the previous example

Window **events** include

onLoad() Message sent each time a document is loaded into a window Can be used to trigger actions (e g , calling a function) Usually placed within the <body>tag, for example <body onLoad="displayWelcome()">

would cause the function displayWelcome() to execute automatically every time the document is loaded or refreshed

onUnload() Message sent each time a document is closed or replaced with another document Can be used to trigger actions (e g , calling a function) Usually placed within the <body> tag, for example <body onUnload="displayFarewell()"> would cause the function displayFarewell() to execute automatically every time the document is closed or refreshed

The Document Object

The Document object represents the HTML document displayed in a browser window It has properties, methods and events that allow the programmer to change the way the document is displayed in response to user actions or other events

Document **properties** include

bgColor The colour of the background For example document bgColor = "lightgreen",

would cause the background colour of the document to change to light-green

fgColor The colour of the text For example document fgColor = "blue",

will cause the colour of the text in the document to change to blue

linkColor The colour used for un-visited links (i e , those that have not yet been clicked-upon by the user) For example document linkColor = "red",

will change the colour of all the un-visited links in a document to red

alinkColor The colour used for an active link (i e , the one that was clicked-upon most recently, or is the process of being clicked) For example document alinkColor = "lightred", will change the colour of active links in a document to light-red

vlinkColor The colour used for visited links (i e , those that have previously been clicked-upon by the user) For example document vlinkColor = "darkred",

will change the colour of all the visited links in a document to dark-red

title The title of the document, as displayed at the top of the browser window For example document title = "This title has been changed", will replace the existing page title with the text "This title has been changed"

forms An array containing all the forms (if any) in the document It accepts an index

number in the following way forms[index-number] where index-number is the number of a particular form Forms are automatically numbered from 0, starting at the beginning of the document, so the first form in an HTML document will always have the index-number 0

Document **methods** include

write() Allows a string of text to be written to the document Can be used to generate new HTML code in response to user actions For example

```
document write("<h1>Hello</h1> "),
document write("<p>Welcome to the new page</p>"),
document write("<p>To return to the lecture notes,"),
document write("<a href='05_JS4 html'>click here </a></p>"),
```

will replace the existing page display with the HTML code contained within the brackets of the document write() methods This code will display the text "Hi, there!" and "Welcome to the new page", followed by a link back to this page

The Form Object

When you create a form in an HTML document using the <form> and </form> tags, you automatically create a form object with properties, methods and events that relate to the form itself and to the individual elements within the form (e g , text boxes, buttons, radio-buttons, etc) Using JavaScript, you can add behaviour to buttons and other form elements and process the information contained in the form

Form **properties** include

name The name of the form, as defined in the HTML <form> tag when the form is created, for example <form name="myForm">
This property can be accessed using JavaScript For example, this paragraph is part of a form that contains the example buttons It is the third form in the document (the others contain the buttons for the Window and Document object examples) To obtain the name of this form, we could use the following code alert(document forms[2] name),

method The method used to submit the information in the form, as defined in the HTML <form> tag when the form is created, for example
<form method="POST">
The method property can be set either to POST or GET (see under 'forms' in any good HTML reference book if you're not sure about the use of the POSTand GET methods)

action — The action to be taken when the form is submitted, as defined in the HTML <form> tag when the form is created, for example <form action="mailto:sales@bigco.com"> The action property specifies either the URL to which the form data should be sent (e g , for processing by a CGI script) or mailto followed by an email address to which the data should be sent (for manual processing by the recipient) See under 'forms' in any good HTML reference book for more information on the use of the action attribute

This property can be accessed using JavaScript For example, the present form has its action attribute set to "mailto sales@bigco com" (even though it's not actually going to be submitted) So the code alert(document forms[2] action),

length — The number of elements (text-boxes, buttons, etc) in the form For example alert(document forms[2] length),

elements — An array of all the elements in the form Individual elements are referenced by index-number

Elements are automatically numbered from 0, starting at the beginning of the form, so the first element in a form will always have the index-number 0 For example alert(document forms[2] elements[0] name),

will display the name of the first element in this form, which is the button labelled "Get Form Name" It's name is "get_form_name"

Form **methods** include

submit() — Submits the form data to the destination specified in the action attribute using the method specified in the method attribute As such it performs exactly the same function as a standard 'submit' button, but it allows the programmer greater flexibility For example, using this method it is possible to create a special-purpose 'submit' button that has more functionality than a standard 'submit' button, perhaps checking the data or performing some other processing before submitting the form

Form **events** include

OnSubmit — Message sent each time a form is submitted Can be used to trigger actions (e g , calling a function) Usually placed within the <form> tags, for example <form onSubmit="displayFarewell()">would cause the function displayFarewell() to execute automatically every time the form is submitted

HD5A

B.C.A., LL.B (Hons.) DEGREE (SEMESTER) EXAMINATIONS,(MODEL PAPER)

(For the candidates admitted from academic year 2015 - 2016 onwards)

THIRD YEAR – FIFTH SEMESTER

OBJECT ORIENTED PROGRAMMING LANGUAGE – JAVA & WEB TECHNOLOGY

Time 2½ hours Maximum 70 marks

PART A – (2 X 12 = 24 marks)

Answer TWO of the following in about 500 words each

1 Explain the features of java

2 What are the different form tags in HTML?

3 Define browser Describe the features of Event handling in DOM

PART B – (2 X 7 = 14 marks)

Answer TWO of the following in about 300 words each

4 Write a program to find a given number is prime number or not

5 Explain operator's associvity in detail (refer pg no 69)

6 Detail out <INPUT> tag with example (refer pg no 146)

PART C – (5 X 4 = 20 marks)

7 Write short notes on FIVE of the following

a) Differentiate between overloading and overriding with example each

b) What is the importance of DNS?

c) Why Hybrid inheritance is not supported by java – justify

d) How frames are used in HTML?

e) What is the use of "final" keyword in Java? (refer pg no 94)

f) Explain heading tags (refer pg no 134)

g) Define types of class reference (refer pg no 20)

PART D – (6 X 2 = 12 marks)

8 Answer SIX of the following very briefly

(a) Define thread

(b) How to set class path?

(c) Webpage

(d) Compare radio button and check box

(e) What is scripting?

(f) Define **break** keyword

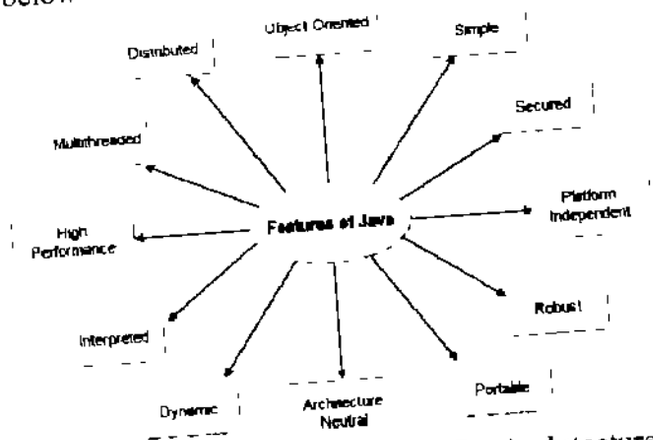
(g) Justify the need for Urls

(h) What do you mean by identifiers?

Answers

1. Features of Java:

There are many features in java They are also known as java buzzwords The Java Features are explained below



1 Simple
2 Object-Oriented
3 Portable
4 Platform independent
5 Secured
6 Robust
7 Architecture neutral
8 Dynamic
9 Interpreted
10 High Performance
11 Multithreaded
12 Distributed

1) Compiled and Interpreter - has both Compiled and Interpreter Feature Program of java is First Compiled and Then it is must to Interpret it First of all The Program of java is Compiled then after Compilation it creates Bytes Codes rather than Machine Language Then After Bytes Codes are Converted into the Machine Language is Converted into the Machine Language with the help of the Interpreter So For Executing the java Program First of all it is necessary to Compile it then it must be Interpieter

2) Platform Independent:- JavaLanguage is Platform Independent means program of java is Easily transferable because after Compilation of java program bytes code will be created then we have to just transfer the Code of Byte Code to another Computer This is not necessary for computers having same Operating System in which the code of the java is Created and Executed After Compilation of the Java Program We easily Convert the Program of the java top the another Computer for Execution

3) Object-Oriented:- We Know that is purely OOP Language that is all the Code of the java Language is Written into the classes and Objects So For This feature java is Most Popular Language because it also Supports Code Reusability, Maintainability etc

4) Robust and Secure:- The Code of java is Robust andMeans ot first checks the reliability of the code before Execution When We trying to Convert the Higher data type into the Lower Then it Checks the Demotion of the Code the It Will Warns a User to Not to do this So it is called as Robust

Secure : When We convert the Code from One Machine to Another the First Check the Code either it is Effected by the Virus or not or it Checks the Safety of the Code if code contains the Virus then it will never Executed that code on to the Machine

5) Distributed:- Java is Distributed Language Means because the program of java is compiled onto one machine can be easily transferred to machine and Executes them on another machine because facility of Bytes Codes So java is Specially designed For Internet Users which uses the Remote Computers For Executing their Programs on local machine after transferring the Programs from Remote Computers or either from the internet

6) Simple Small and Familiar:- is a simple Language Because it contains many features of other Languages like c and C++ and Java Removes Complexity because it doesn't use pointers, Storage Classes and Go to Statements and java Doesn't support Multiple Inheritance

7) Multithreaded and Interactive:- Java uses Multithreaded Techniques For Execution Means Like in other in Structure Languages Code is Divided into the Small Parts Like These Code of java is divided into the Smaller parts those are Executed by java in Sequence and Timing Manner this is Called as Multithreaded In this Program of java is divided into the Small parts those are Executed by Compiler of java itself Java is Called as Interactive because Code of java Supports Also CUI and Also GUI Programs

8) Dynamic and Extensible Code:- Java has Dynamic and Extensible Code Means With the Help of OOPS java Provides Inheritance and With the Help of Inheritance we Reuse the Code that is Pre-defined and Also uses all the built in Functions of java and Classes

9) Distributed:- Java is a distributed language which means that the program can be design to run on computer networks Java provides an extensive library of classes for communicating ,using TCP/IP protocols such as HTTP and FTP This makes creating network connections much easier than in C/C++ You can read and write objects on the remote sites via URL with the same ease that programmers are used to when read and write data from and to a file This helps the programmers at remote locations to work together on the same project

10) Secure: Java was designed with security in mind As Java is intended to be used in networked/distributor environments so it implements several security mechanisms to protect you against malicious code that might try to invade your file system For example. The absence of pointers in Java makes it impossible for applications to gain access to memory locations without proper authorization as memory allocation and referencing model is completely opaque to the programmer and controlled entirely by the underlying run-time platform

11) Architectural Neutral One of the key features of Java that makes it different from other programming languages is architectural neutral (or platform independent) This means that the programs written on one platform can run on any other platform without having to rewrite or recompile them In other words, it follows 'Write-once-run-anywhere' approach

Java programs are compiled into byte-code format which does not depend on any machine architecture but can be easily translated into a specific machine by a Java Virtual Machine (JVM) for that machine This is a significant advantage when developing applets or applications that are downloaded from the Internet and are needed to run on different systems

12) Portable: The portability actually comes from architecture-neutrality In C/C++, source code may run slightly differently on different hardware platforms because of how these platforms implement arithmetic operations In Java, it has been simplified

Unlike C/C++, in Java the size of the primitive data types are machine independent For example, an int in Java is always a 32-bit integer, and float is always a 32-bit IEEE 754 floating point number. These consistencies make Java programs portable among different platforms such as Windows, Unix and Mac

13) Interpreted Unlike most of the programming languages which are either complied or interpreted, Java is both complied and interpreted The Java compiler translates a java source file to bytecodes and the Java interpreter executes the translated byte codes directly on the system that implements the Java Virtual Machine These two steps of compilation and interpretation allow extensive code checking and improved security

14) High performance: Java programs are complied with portable intermediate form known as bytecodes, rather than to native machine level instructions and JVM executes Java bytecode on any machine on which it is installed This architecture means that Java programs are faster than program or scripts written in purely interpreted languages but slower than C and C++ programs that compiled to native machine languages Although in the early releases of Java, the interpretation of by bytecode resulted in slow performance but the advance version of JVM uses the adaptive and Just in time (JIT) compilation technique that improves performance by converting Java bytecodes to native machine instructions on the fly

2 **Definition and Usage:** The <form> tag is used to create an HTML form for user input The <form> element can contain one or more of the following form elements (refer pg no 145)

- <input>
- <textarea>
- <button>
- <select>
- <option>
- <optgroup>
- <fieldset>
- <label>

3 **Browser:** An internet browser, also known as a web browser or simply a browser, is a software program that you use to access the internet and view web pages on your computer The main purpose of an internet browser is to translate, or render, the code that websites are designed-in into the text, graphics, and other features of the web pages

DOM Events: HTML DOM events allow JavaScript to register different event handlers on elements in an HTML document Events are normally used in combination with functions, and the function will not be executed before the event occurs (such as when a user clicks a button)

Mouse Events

Event	Description
onclick	The event occurs when the user clicks on an element
oncontextmenu	The event occurs when the user right-clicks on an element to open a context menu
ondblclick	The event occurs when the user double-clicks on an element
onmousedown	The event occurs when the user presses a mouse button over an element
onmouseenter	The event occurs when the pointer is moved onto an element

onmouseleave	The event occurs when the pointer is moved out of an element
onmousemove	The event occurs when the pointer is moving while it is over an element
onmouseover	The event occurs when the pointer is moved onto an element, or onto one of its children
onmouseout	The event occurs when a user moves the mouse pointer out of an element, or out of one of its children
onmouseup	The event occurs when a user releases a mouse button over an element

Keyboard Events

Event	Description
onkeydown	The event occurs when the user is pressing a key
onkeypress	The event occurs when the user presses a key
onkeyup	The event occurs when the user releases a key

Frame/Object Events

Event	Description
onabort	The event occurs when the loading of a resource has been aborted
onbeforeunload	The event occurs before the document is about to be unloaded
onerror	The event occurs when an error occurs while loading an external file
onhashchange	The event occurs when there has been changes to the anchor part of a URL
onload	The event occurs when an object has loaded
onpageshow	The event occurs when the user navigates to a webpage
onpagehide	The event occurs when the user navigates away from a webpage
onresize	The event occurs when the document view is resized
onscroll	The event occurs when an element's scrollbar is being scrolled
onunload	The event occurs once a page has unloaded (for <body>)

Form Events

Event	Description
onblur	The event occurs when an element loses focus
onchange	The event occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <keygen>, <select>, and <textarea>)
onfocus	The event occurs when an element gets focus

onfocusin	The event occurs when an element is about to get focus
onfocusout	The event occurs when an element is about to lose focus
oninput	The event occurs when an element gets user input
oninvalid	The event occurs when an element is invalid
onreset	The event occurs when a form is reset
onsearch	The event occurs when the user writes something in a search field (for <input="search">)
onselect	The event occurs after the user selects some text (for <input> and <textarea>)
onsubmit	The event occurs when a form is submitted

4
```
/* Java Program Example - Check Prime or Not */
import java util Scanner,
public class JavaProgram
{
  public static void main(String args[])
  {
    int num, i, count=0,
    Scanner scan = new Scanner(System in),
    System out print("Enter a Number  "),
    num = scan nextInt(),
    for(i=2, i<num, i++)
    {
      if(num%i == 0)
      {
        count++,
        break,
      }
    }
    if(count == 0)
    {
      System out print("This is a Prime Number"),
    }
    else
    {
      System out print("This is not a Prime Number"),
    }
  }
}
```

7 (a) **Overloading vs Overriding in Java**

- Overloading happens at compile-time while Overriding happens at runtime The binding of overloaded method call to its definition has happens at compile-time however binding of overridden method call to its definition happens at runtime
- Static methods can be overloaded which means a class can have more than one static method of same name Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class
- The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required Overriding is all about giving a specific implementation to the inherited method of parent class
- Static binding is being used for overloaded methods and dynamic bindingis being used for overridden/overriding methods

(b) DNS is the de facto way of addressing computer services We use names more often than we use IP addresses If DNS were to fail we would have no way of finding the IP address, and would be unable to access any Internet services DNS also virtualizes a service, making its IP address less significant If the service changes IP address but retains the same name, nobody is likely to notice DNS makes it easier to move, add and change Internet services DNS resolves more than just IP addresses That system is extensible to multiple record types For example the A record takes a name and returns a v4 IP AAAA takes a name and returns a v6 IP MX takes a name and returns an ordered list of servers to accept mail for that domain SRV and TXT records are just textual fields that have been adapted for any number of things The PTR record takes an IP address and returns a name, sometimes The NS record takes a name and returns a list of servers that are knowledgeable about the records in that domain, where subqueries should be directed

(c) yes, Hybrid inheritance is the combination of every type of inheritance that exist As java doesn't support multiple inheritance, hybrid inheritance also can't be implemented Similarly, as multiple inheritance is implemented through interfaces, hybrid inheritance can be implemented with the help of interface

(d) HTML frames are used to divide your browser window into multiple sections where each section can load a separate HTML document A collection of frames in the browser window is known as a frameset The window is divided into frames in a similar way the tables are organized· into rows and columns

Disadvantages of Frames

There are few drawbacks with using frames, so it's never recommended to use frames in your webpages –

- Some smaller devices cannot cope with frames often because their screen is not big enough to be divided up
- Sometimes your page will be displayed differently on different computers due to different screen resolution
- The browser's back button might not work as the user hopes
- There are still few browsers that do not support frame technology

8 (a) Thread a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process

(b) Open Control Panel and Select System and Security

- Select System
- Select Advanced System Settings
- Select Environment Variables
- Select and Edit Path Environment variable and click "ok"

(c) A web page is a document commonly written in HyperText Markup Language (HTML) that is accessible through the Internet or other network using an Internet browser A web page is accessed by entering a URL address and may contain text, graphics, and hyperlinks to other web pages and files The page you are reading now is an example of a web page

(d) Radio buttons are used when there is a list of two or more options that are mutually exclusive and the user must select exactly one choice In other words, clicking a non-selected radio button will deselect whatever other button was previously selected in the list

Checkboxes are used when there are lists of options and the user may select any number of choices, including zero, one, or several In other words, each checkbox is independent of all other checkboxes in the list, so checking one box doesn't uncheck the others

(e) Scripting languages, which can be embedded within HTML, commonly are used to add functionality to a Web page, such as different menu styles or graphic displays or to serve dynamic advertisements These types of languages are client-side scripting languages, affecting the data that the end user sees in a browser window Other scripting languages are server-side scripting languages that manipulate the data, usually in a database, on the server

(f) Break The break statement in Java programming language has the following two usages –

- When the break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop
- It can be used to terminate a case in the switch statement

(g) A URL (Uniform Resource Locator), as the name suggests, provides a way to locate a resource on the web, the hypertext system that operates over the internet The URL contains the name of the protocol to be used to access the resource and a resource name The first part of a URL identifies what protocol to use The second part identifies the IP address or domain name where the resource is located

(h) *Identifiers* are the names of variables, methods, classes, packages and interfaces Unlike literals they are not the things themselves, just ways of referring to them In the HelloWorld program, HelloWorld, String, args, main and println are identifiers Identifiers must be composed of letters, numbers, the underscore _ and the dollar sign $ Identifiers may only begin with a letter, the underscore or a dollar sign