DBATU SUMMER 2022 JAVA PAPER SOLVED

Q.1 Solve any Two the Following

A) What are the primitive data types in java? Write about type conversions.

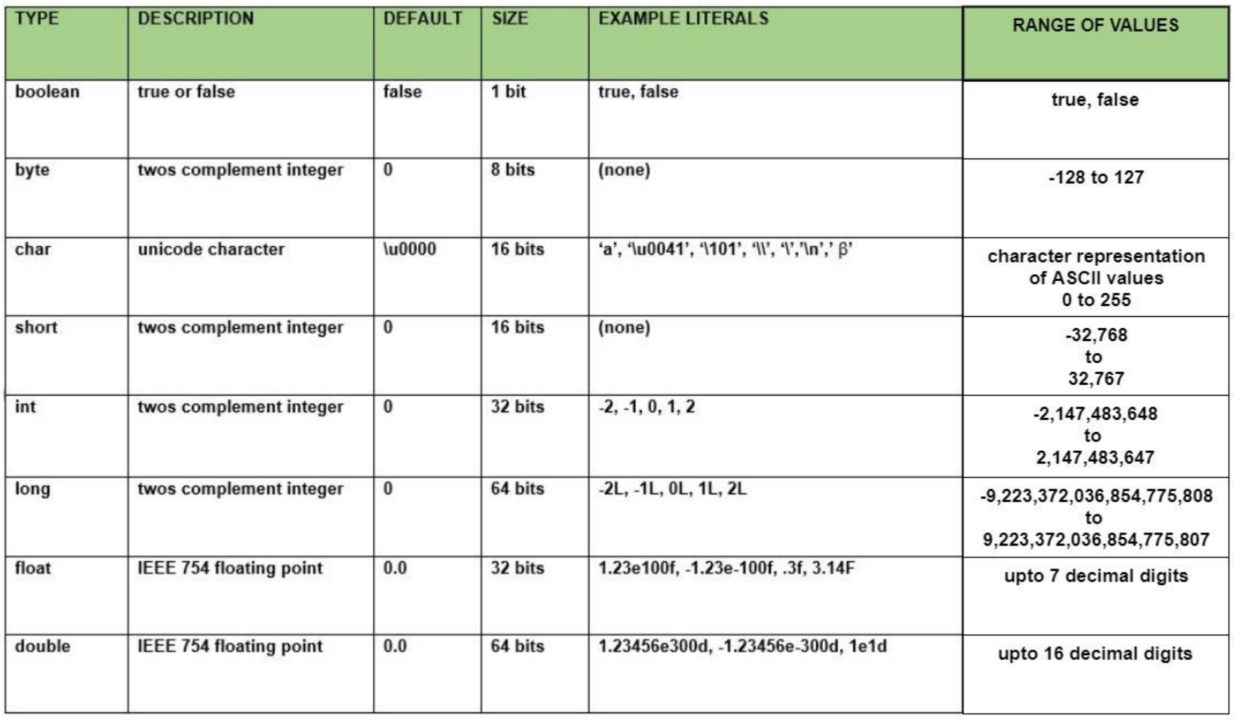
ANS: Data Types in Java:

In java, data types are classified into two categories:

1. Primitive Data type or built-in data type
2. Non-Primitive Data type or derived data type

- Primitive Data type or built-in data type

- In Java, the primitive data types are the predefined data types of Java.
- They specify the size and type of any standard values.
- There are 8 types of primitive data types:
  - Boolean data type
  - byte data type
  - char data type
  - short data type
  - int data type
  - long data type
  - float data type
  - double data type



TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\','\n',' β'	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

- Type Conversion:

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

- Types of Type Casting:

There are two types of type casting:

- Widening Type Casting
- Narrowing Type Casting

▪ Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically

Program Ex:

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
long y = x;
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Output:

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

- Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

Program Ex:

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
long l = (long)d;
int i = (int)l;
System.out.println("Before conversion: "+d);
System.out.println("After conversion into long type: "+l);
System.out.println("After conversion into int type: "+i);
}
}
```

Output:

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

B) What is nested class? Differentiate between static nested classes and non-static nested classes.

ANS:

- In Java, it is possible to define a class within another class, such classes are known as nested classes.
- They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation, and creates more readable and maintainable code.

- The scope of a nested class is bounded by the scope of its enclosing class. Thus in below example, class NestedClass does not exist independently of class OuterClass.
- A nested class has access to the members, including private members, of the class in which it is nested. But the enclosing class does not have access to the member of the nested class.
- A nested class is also a member of its enclosing class.
- As a member of its enclosing class, a nested class can be declared private, public, protected, or package private(default).
- Nested classes are divided into two categories:
  1. **static nested class**: Nested classes that are declared static are called static nested classes.
  2. **inner class:** An inner class is a non-static nested class.

Syntax:

```
class OuterClass
{
...
   class NestedClass
   {
       ...
   }
}
```

Difference between static nested classes and non-static nested classes:

Static Nested Classes	**Non-Static Nested Classes**
Without an outer class object existing, there may be a static nested class object. That is, static nested class object is not associated with the outer class object.	Without an outer class object existing, there cannot be an inner class object. That is, the inner class object is always associated with the outer class object.
Inside static nested class, static members can be declared.	Inside normal/regular inner class, static members can’t be declared.
As main() method can be declared, the static nested class can be invoked directly from the command prompt.	As main() method can’t be declared, regular inner class can’t be invoked directly from the command prompt.
Only a static member of outer class can be accessed directly.	Both static and non static members of outer class can be accessed directly.

C) What is a constructor? What is its requirement in programming? Explain with program.

ANS:

*** Constructor**

- A constructor in Java is a **special method** that is used to initialize objects.
- The constructor is called when an object of a class is created.
- It can be used to set initial values for object attributes.
- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created.
- At the time of calling the constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- Constructors must have the same name as the class.

Types of Constructors:

- No-argument constructor (Default Constructor)
- Parameterized Constructor

No-argument constructor (Default Constructor):

- A constructor that has no parameter is known as the No-argument or Zero argument constructor.
- If we don't define a constructor in a class, then the compiler creates a **constructor(with no arguments)** for the class.

And if we write a constructor with arguments or no arguments then the compiler does not create a default constructor.

Syntax:

```
<class_name>(){}
```

Parameterized Constructor:

- A constructor that has parameters is known as parameterized constructor.
- If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Why is Constructor required in programming ?

Program:

```
public class Person {
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public Person() {
    this.name = "Unknown";
    this.age = 0;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}

public class Main {
  public static void main(String[] args) {
    Person person1 = new Person("John Doe", 30);
    Person person2 = new Person();
    System.out.println(person1.getName() + " is " + person1.getAge() + " years old.");
    System.out.println(person2.getName() + " is " + person2.getAge() + " years old.");
  }
}
```

Output:

```
John Doe is 30 years old.
Unknown is 0 years old.
```

Q.2 Solve any two the following

A) Write a program to implement the Fibonacci series using for loop control structure.

ANS:

Program:

```
class FibonacciSeries {
        static int fib(int n){
                if (n <= 1){
                        return n;
                }
                return fib(n - 1)+ fib(n - 2);
        }

        public static void main(String args[]){
                int n = 10;
                for (int i = 0; i < n; i++) {
                        System.out.print(fib(i) + " ");
                }
        }
}
```

Output:

```
        Fibonacci Series:
        1 1 2 3 5 8 13 21 34 55
```

B) Explain the various non-access modifiers used in java.

ANS:

Java provides a number of non-access modifiers to achieve many other functionalities.

Non-access modifiers:

i. static
ii. final
iii. abstract

static keyword:

- Static field & static method:
- In a class individual, differentiate, properties are mapped with the help of instance variables, when we need to map some properties at that time we use static keyword
- It is used for memory management

- Static variables:
- Variables declared with static keyword is called static variable
- With the help of static variable, we declare common program, which has some value for all objects.
- It can be used to refer common properties
- It gets memory only once in the class

final keyword:

- The final keyword in java is used to restrict the user.
- Final variable:
- If you make any variable as final, you cannot change the value of final variable(It will be constant).

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

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

abstract keyword:

- The abstract keyword is used to achieve abstraction in Java.

Syntax of abstract:

```
abstract class ClassName  {
      abstract void methodName();
}
```

C) How to define a package? How to access, import a package? Explain with examples.

ANS:

Java Packages:

- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- The keyword "**import**" is used to import java packages.
- To declare a package use "**package**" keyword.

There are two types of packages in java:

1. User-defined packages
2. Built-in packages

Syntax: To access/ import a package

```
import package.name.*;
```

Syntax: To define a package

```
package package_name;
```

Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

Program:

```
// File: com/example/mypackage/MyClass.java
package com.example.mypackage;

public class MyClass {
    public void myMethod() {
        System.out.println("Hello, world!");
    }
}
```

```
// File: Main.java
import com.example.mypackage.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.myMethod();
    }
}
```

Output:

```
Hello, world!
```

Q.3 Solve any two the following

A) How to declare and create array in JAVA? How to pass an array to method?

ANS:

Arrays:

- An array is a container that holds fixed number of values of similar data type
- In java all arrays are dynamically allocated.
- Arrays are stored in contagious memory
- Java array variable can be declared like other variable with [] after the data type.

To declare and create an array in Java, you can follow the **syntax** below:

```
data_type[] array_name = new data_type[array_size];
```

For example, to create an array of integers with 5 elements, you would write:

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

Passing array as a parameter to a method:

- We can pass array as a parameter to a method.
- The array which we are passing to a method as a parameter can be single dimensional or Multi-Dimensional

To pass an array to a method in Java, you can include the array as a parameter when defining the method.

The syntax for passing an array to a method is as follows:

```
public static void methodName(data_type[] array_name) {
    // method code
}
```

For example, if you want to pass the myArray array that we declared earlier to a method called myMethod, you would write:

```
public static void myMethod(int[] myArray) {
    // method code
}
```

Then, you can call this method and pass the myArray array as an argument like this:

```
myMethod(myArray);
```

Program:

```
public class OneDArray {
  void show(int x[ ]) {
              for(int i = 0; i < x.length; i++)  {
                System.out.print(x[i]+ " ");
              }
               }
}
public class OneDArraytoMethod {
public static void main(String[ ] args) {
    int[ ] x = {2, 3, 4, 5, 6, 7, 8};
     OneDArray obj = new OneDArray();
     System.out.println("Value of array x: ");
      obj.show(x);
  }
}
```

Output:

```
Value of array x:
2 3 4 5 6 7 8
```

B) What is variable length argument, explain with program.

ANS: Variable Argument (Varargs):

- The varrags allows the method to accept zero or muliple arguments.
- A variable-length argument is a feature that allows a function to receive any number of arguments.
- Variable length argument lists make it possible to write a method that accepts any number of arguments when it is called.
- Because of these users gets freedom to pass any number of arguments to a method.

Syntax:

return_type method_name(data_type… variableName){}

Program:

```
public class VarargsExample {

  public static void printNumbers(int... numbers) {
    System.out.print("Numbers: ");
    for (int number : numbers) {
      System.out.print(number + " ");
    }
    System.out.println();
  }

  public static void main(String[] args) {
    printNumbers(1, 2, 3);
    printNumbers(4, 5);
    printNumbers(6);
  }
}
```

Output:

```
Numbers: 1 2 3
Numbers: 4 5
Numbers: 6
```

C) Write a java program to sort a numeric array in ascending order

ANS:

Program:

```
public class SortAsc {
    public static void main(String[] args) {
        int [] arr = new int [] {5, 2, 8, 7, 1};
        int temp = 0;

        System.out.println("Elements of original array: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }

        for (int i = 0; i < arr.length; i++) {
            for (int j = i+1; j < arr.length; j++) {
                if(arr[i] > arr[j]) {
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
        }

        System.out.println();

        System.out.println("Elements of array sorted in ascending order: ");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Output:

```
Elements of original array:
5 2 8 7 1
Elements of array sorted in ascending order:
1 2 5 7 8
```

Alternate Program:

```
import java.util.Arrays;

public class SortArray {
    public static void main(String[] args) {
        int[] arr = { 5, 2, 8, 1, 9 };
        Arrays.sort(arr);
        System.out.println("Sorted Array: " + Arrays.toString(arr));
    }
}
```

Output:

```
Sorted Array: [1, 2, 5, 8, 9]
```

Q.4 **Solve any two the following**

A) What are the benefits of inheritance? Explain the various forms of inheritance with suitable code segments.

ANS:

Inheritance:

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviour of a parent object
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.
- The benefits of inheritance include code reusability, modularity, and extensibility.

Code reusability: Inheritance allows you to reuse code that has already been written in the base class. This can save time and effort, as you don't have to rewrite the same code again in the derived class.

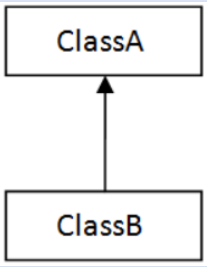
Modularity: Inheritance promotes modularity by allowing you to separate different functionalities into separate classes. This makes your code more organized and easier to maintain.
Extensibility: Inheritance allows you to extend the functionality of an existing class by creating a derived class that adds new methods and properties. This can help you to adapt to changing requirements and add new features to your application without having to modify existing code.

Types of Inheritance:

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

Single Inheritance:

- When a class inherits another class, it is known as a *single inheritance*.



Program:

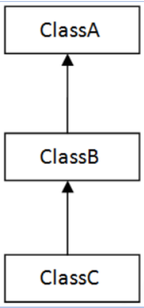
```
    class Animal {
        public void eat() {
            System.out.println("I am eating");
        }
    }

    class Dog extends Animal {
        public void bark() {
            System.out.println("Woof!");
        }
    }

    public class Main {
        public static void main(String[] args) {
            Dog dog = new Dog();
            dog.eat();
            dog.bark();
        }
}
```

Output:

```
I am eating
Woof!
```

Multi-level Inheritance:

- When there is a chain of inheritance, it is known as *multilevel inheritance*.
- In multi-level inheritance, a derived class extends a base class, and another derived class extends the first derived class.



Program:

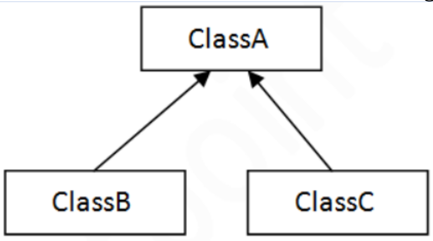
```
class Animal {
    public void eat() {
        System.out.println("I am eating");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Woof!");
    }
}

class Bulldog extends Dog {
    public void run() {
        System.out.println("I am running");
    }
}

public class Main {
    public static void main(String[] args) {
        Bulldog bulldog = new Bulldog();
        bulldog.eat();
        bulldog.bark();
        bulldog.run();
    }
}
```

Output:

```
I am eating
Woof!
I am running
```

Hierarchical Inheritance:

When two or more classes inherits a single class, it is known as *hierarchical inheritance*.



Program:

```
class Animal {
    public void eat() {
        System.out.println("I am eating");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    public void meow() {
        System.out.println("Meow!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // outputs "I am eating"
        dog.bark(); // outputs "Woof!"

        Cat cat = new Cat();
        cat.eat(); // outputs "I am eating"
        cat.meow(); // outputs "Meow!"
    }
}
```

Output:

```
I am eating
Woof!
I am eating
Meow!
```

B) Describe the uses of final and super keyword with respect to inheritance

ANS:

final: When a method, class, or variable is declared as final, it means that it cannot be overridden or modified by any subclass. This is useful when you want to prevent any further changes to a class or method that is critical to the functioning of the program. For example, a final class may represent a utility class that should no be extended, or a final method may represent a core functionality of a class that should not be altered.

Ex:

```
public final class MyFinalClass {
    // ...
}

public class MySubclass extends MyFinalClass {
    // This will give a compile-time error because MyFinalClass is final
}
```

In this example, the MyFinalClass is declared as final, which means that it cannot be subclassed.
When we try to create a subclass MySubclass of MyFinalClass, the Java compiler will give a compile-time error.

Super:

- The super keyword is used to call the superclass's methods, constructors, or variables from the subclass.
- This is useful when you want to access the functionalities of the superclass that may not be available in the subclass.
- For example, if a subclass overrides a method from the superclass, it may still want to call the original implementation of the method using super.method().
- Similarly, if a subclass needs to access a variable or constructor from the superclass, it can use super.variable or super() respectively

Ex:

```
public class Vehicle {
    protected int speed;

    public Vehicle(int speed) {
        this.speed = speed;
    }

    public void accelerate() {
        speed += 10;
    }
}

public class Car extends Vehicle {
    private int gear;

    public Car(int speed, int gear) {
        super(speed);
        this.gear = gear;
    }

    public void shiftUp() {
        gear++;
    }

    public void accelerate() {
        super.accelerate();
        speed += 20;
    }
}
```

Output:

C) Design an interface called shape with method draw() and getarea(). Further design two classes called Circle and Rectangle that implements Shape interface to compute area of respective shapes. Use appropriate getter and setter methods. Write a java program for the same.

ANS:

Program:

```
interface Shape {
    void draw();
    double getArea();
}

class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing Circle...");
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle implements Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double getLength() {
        return length;
    }

    public void setLength(double length) {
        this.length = length;
    }

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }
    @Override
    public void draw() {
        System.out.println("Drawing Rectangle...");
    }

    @Override
    public double getArea() {
        return length * width;
    }
}

class Main {
    public static void main(String[] args) {
        Shape circle = new Circle(5.0);
        circle.draw();
        System.out.println("Area of Circle: " + circle.getArea());

        Shape rectangle = new Rectangle(4.0, 6.0);
        rectangle.draw();
        System.out.println("Area of Rectangle: " + rectangle.getArea());
    }
}
```

Output:

```
Drawing Circle...
Area of Circle: 78.53981633974483Drawing Rectangle...
Area of Rectangle: 24.0
```

Q.5 Solve any two the following

A) How to handle multiple catch blocks for a nested try block? Explain with an example

ANS:

- In Java, when you have nested try blocks, you can handle exceptions using multiple catch blocks.
- Each catch block will handle exceptions thrown by the try block nested within it. Here's an example:

Program:

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            try {
                int[] arr = new int[5];
                arr[6] = 8;
            }
catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out of bounds exception\n ");
            }
            int a = 5/0;
        }
catch (ArithmeticException e) {
            System.out.println("Arithmetic exception, can't divide by zero ");
        } catch (Exception e) {
            System.out.println("Generic exception");
        }
    }
}
```

Output:

```
Array index out of bounds exception
Arithmetic exception can't divide by zero
```

- In this example, we have a nested try block. The inner try block initializes an array of 5 integers and then tries to access the 6th index, which throws an ArrayIndexOutOfBoundsException. This exception is caught by the inner catch block, which prints an appropriate message to the console.
- After the inner try-catch block, we have an arithmetic expression that attempts to divide 5 by 0, which throws an ArithmeticException. This exception is caught by the first catch block outside the inner try-catch block, which prints a message to the console.

B) Write a java program to create own exception for negative value exception if the user enter negative value

ANS:

Program:

```
import java.util.Scanner;

class NegativeValueException extends Exception {
    public NegativeValueException() {
        super("Negative values are not allowed.");
    }
}

class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int value;

        try {
            System.out.print("Enter a positive integer: ");
            value = scanner.nextInt();

            if (value < 0) {
                throw new NegativeValueException();
            }

            System.out.println("The value you entered is: " + value);
        } catch (NegativeValueException ex) {
            System.out.println(ex.getMessage());
        } catch (Exception ex) {
            System.out.println("Invalid input.");
        }
    }
}
```

Output:

```
Enter a positive integer: 10
The value you entered is: 10
```

```
Enter a positive integer: -5
Negative values are not allowed.
```

C)	Explain pattern matching. What is the different meta characters used in pattern matching?
ANS:	

Pattern matching is a technique used in computer science to find specific patterns within strings of text. In Java, pattern matching is implemented through the use of regular expressions, which are sequences of characters that define a search pattern.

Regular expressions in Java can include various meta characters that are used to specify different types of patterns. Here are some of the most common meta characters used in Java pattern matching:

1. Dot (.) - Matches any character except for a newline.
2. Asterisk (*) - Matches zero or more occurrences of the preceding character or group.
3. Plus (+) - Matches one or more occurrences of the preceding character or group.
4. Question mark (?) - Matches zero or one occurrence of the preceding character or group.
5. Backslash () - Used to escape special characters or indicate special sequences.
6. Brackets ([ ]) - Used to specify a set of characters that can match any single character.
7. Pipe (|) - Used to specify a set of alternatives.
8. Caret (^) - Matches the beginning of the input string.
9. Dollar sign ($) - Matches the end of the input string.

These meta characters can be combined in various ways to create complex search patterns that can match specific patterns within strings. Java provides the **java.util.regex** package, which includes classes like **Pattern** and **Matcher** that allow developers to work with regular expressions and perform pattern matching operations on strings.

*** * * End * * ***